

**Dal catalogo Apogeo
Informatica**

- Biermann, Ramm, *Le idee dell'informatica*
Bolchini, Brandolesi, Salice, Sciuto, *Reti logiche*, 2a edizione
Coppola, Mizzaro, *Laboratorio di programmazione in Java*
Bruni, Corradini, Gervasi, *Programmazione in Java*
Deitel, C Corso completo di programmazione, 3a edizione
Deitel, C++ Fondamenti di programmazione, 2a edizione
Deitel, C++ Tecniche avanzate di programmazione, 2a edizione
Deitel, Java Fondamenti di programmazione, 3a edizione
Deitel, Java Tecniche avanzate di programmazione, 3a edizione
Della Mea, Di Gaspero, Scagnetto, *Programmazione web lato server*
Hennessy, Patterson, *Architettura degli elaboratori*
Horstmann, *Concetti di informatica e fondamenti di Java*, 4a edizione
Horstmann, *Progettazione del software e design pattern in Java*
Laganà, Righi, Romani, *Informatica. Concetti e sperimentazioni*, 2a edizione
Mazzanti, Milanese, *Programmazione di applicazioni grafiche in Java*
Peterson, Davie, *Reti di calcolatori*, 2a edizione
Pigni, Ravarini, Sciuto, *Sistemi per la gestione dell'informazione*, 2a edizione
Schneider, Gersting, *Informatica*

Programmazione in C

Kim N. King

Edizione italiana a cura di
Andrea Schaerf

APOGEO

Alcuni esercizi richiedono risposte non ovvie (alcuni le definirebbero "domande difficili"). Dato che i programmi C contengono spesso numerosi esempi di questo tipo di codice, penso sia necessario fornire un po' di pratica a riguardo. Tuttavia, sono stato corretto segnalando questi esercizi con un asterisco (*): quando affrontate un esercizio di questo tipo fate molta attenzione e ragionate approfonditamente, oppure è meglio se lo evitate del tutto.

Errori, dimenticanze

Ho fatto un grande sforzo per assicurare l'accuratezza di questo testo. Inevitabilmente, però, ogni libro di queste dimensioni contiene qualche errore. Se ne individuate vi prego di contattarmi presso cbook@knking.com. Inoltre apprezzo qualsiasi opinione sulle caratteristiche del libro che avete trovato più interessanti, su quelle delle quali avreste fatto a meno e su quelle che avreste voluto fossero state aggiunte.

Ringraziamenti

Per prima cosa vorrei ringraziare i miei editor alla Norton: Fred McFarland e Aaron Javicas. Fred è stato coinvolto nella seconda edizione dal principio, mentre Aaron è intervenuto con rapida efficienza per portarla a compimento. Vorrei ringraziare anche il caporedattore Kim Yi, la redattrice Mary Kelly, il responsabile di produzione Roy Tedoff e l'assistente editoriale Carly Fraser.

Sono profondamente in debito con i seguenti colleghi, che hanno rivisto alcuni o tutti i manoscritti della seconda edizione:

Markus Bussmann, dell'Università di Toronto
 Jim Clarke, dell'Università di Toronto
 Karen Reid, dell'Università di Toronto
 Peter Seebach, moderatore di *comp.lang.c.moderated*

Jim e Peter meritano uno speciale riconoscimento per le loro revisioni dettagliate, che mi hanno evitato un buon numero di errori imbarazzanti. I revisori della prima edizione erano, in ordine alfabetico: Susan Anderson-Freed, Manuel E. Bermudez, Lisa J. Brown, Steven C. Cater, Patrick Harrison, Brian Harvey, Henry H. Leitner, Darrel Long, Arthur B. Maccabe, Carolyn Rosner e Patrick Terry.

Ho ricevuto molti commenti utili dai lettori della prima edizione: voglio ringraziare tutti quelli che mi hanno scritto. Anche gli studenti e i colleghi della Georgia State University hanno fornito un prezioso feedback. Ed Bullwinkel e sua moglie Nancy sono stati così gentili da leggere la maggior parte del manoscritto. Sono particolarmente grato al mio capo dipartimento, Yi Pan, che ha supportato il progetto.

Mia moglie, Susan Cole, è stato un pilastro di forza come sempre. Anche i nostri gatti, Dennis, Pounce e Tex hanno contribuito al completamento del libro: le loro occasionali lotte feline mi hanno aiutato a rimanere sveglio quando lavoravo fino a notte fonda.

Infine vorrei ringraziare Alan J. Perlis, che non è più tra noi. Ho avuto il privilegio di studiare brevemente sotto la sua guida a Yale nella metà degli anni Settanta.

1 Introduzione al C

Che cos'è il C? La risposta semplice – un linguaggio di programmazione ampiamente utilizzato che è stato sviluppato nei primi anni '70 presso i laboratori Bell – rende poco l'idea delle speciali caratteristiche del C. Prima di immergervi nei dettagli del linguaggio, diamo uno sguardo alle sue origini, ai suoi scopi e a come è cambiato nel corso degli anni (Sezione 1.1). Discuteremo anche dei suoi punti di forza e delle sue debolezze e vedremo come ricavare il massimo da questo linguaggio (Sezione 1.2).

1.1 Storia del C

Vediamo ora, in breve, la storia del C, dalle sue origini al raggiungimento della maturità come linguaggio standardizzato, fino alle sue influenze sui recenti linguaggi.

Origini

Il C è un sottoprodotto del sistema operativo UNIX che è stato sviluppato presso i laboratori Bell da Ken Thompson, Dennis Ritchie ed altri. Thompson fu l'unico autore della versione originale di UNIX che funzionava sul computer DEC PDP-7 uno dei primi minicalcolatori con solo 8K words di memoria principale (era il 1969 dopo tutto!).

Come tutti gli altri sistemi operativi del tempo, UNIX venne scritto in linguaggio assembly. I programmi scritti con il linguaggio assembly sono solitamente faticosi da gestire nelle fasi di debug e risultano particolarmente difficili da migliorare. UNIX non faceva eccezione a questa regola. Thompson decise che per un'ulteriore sviluppo di UNIX era necessario un linguaggio di livello superiore, e creò un piccolo linguaggio chiamato B. Thompson basò il B su BCPL, un linguaggio di programmazione di sistema sviluppato nella metà degli anni '60. BCPL, a sua volta, ritrovava le sue origini in Algol 60, uno dei primi (e più importanti) linguaggi di programmazione.

Ritchie prese parte al progetto UNIX e iniziò a programmare in B. Nel 1970 i laboratori Bell acquisirono un computer PDP-11 per il progetto e, quando B divenne operativo su tale sistema, Thompson riscrisse una porzione di UNIX in B. A partire dal 1971 divenne evidente quanto il B non fosse adatto al PDP-11, fu così che Ritchie iniziò lo sviluppo di una versione estesa del linguaggio. Inizialmente chiamò il nuovo

linguaggio NB ("New B") ma successivamente, a mano a mano che le divergenze dal B si facevano più evidenti, Ritchie cambiò il nome in C. Il nuovo linguaggio diventò sufficientemente stabile entro il 1973, tanto che UNIX venne riscritto in C. La transizione al C produsse un importante beneficio: la portabilità. Scrivendo compilatori C per gli altri computer presenti nei laboratori Bell, il team di sviluppatori poté far funzionare UNIX anche su tutte quelle macchine.

Standardizzazione

Il C continuò a evolvere durante gli anni '70, specialmente tra il 1977 ed il 1979 e in questo periodo fu stampato il primo libro sul C. *The C Programming Language*, scritto da Brian Kernigan e Dennis Ritchie, pubblicato nel 1978, diventò in poco tempo la bibbia dei programmatore C. Nell'assenza di uno standard ufficiale per il C, questo libro – conosciuto come K&R o "the White Book" per gli affezionati – servì come uno standard *de facto*.

Durante gli anni '70 vi erano relativamente pochi programmatore C, la maggior parte dei quali, peraltro, erano utenti UNIX. Nel 1980 invece, C si era espanso ben oltre gli stretti confini del mondo UNIX. Compilatori C divennero disponibili su una larga varietà di calcolatori funzionanti con differenti sistemi operativi. In particolare, il C iniziò a stabilirsi sulla piattaforma PC IBM che ai tempi stava conoscendo un forte sviluppo.

Con l'aumento della popolarità del C iniziarono i problemi. I programmatore che scrivevano nuovi compilatori C si basavano sul K&R come riferimento, ma sfortunatamente il K&R era approssimativo su alcune caratteristiche del linguaggio. Fu così che differenti compilatori trattarono queste caratteristiche in modo diverso. In più, il K&R non riusciva a fornire una chiara distinzione tra le caratteristiche proprie del C e quelle appartenenti a UNIX. A peggiorare le cose fu il fatto che il C continuò a cambiare, anche dopo la pubblicazione del K&R, attraverso l'aggiunta di nuove caratteristiche e con la rimozione di alcune di quelle preesistenti. La necessità di una descrizione del linguaggio precisa, accurata e aggiornata divenne subito evidente. In assenza di uno standard, infatti, sarebbero nati numerosi dialetti che avrebbero minacciato la portabilità dei programmi C, e quindi, uno dei maggiori punti di forza del linguaggio.

Lo sviluppo di uno standard statunitense per il C iniziò nel 1983 sotto l'egida dell'American National Standard Institute (ANSI). Dopo molte revisioni, lo standard venne completato nel 1988 e formalmente approvato nel dicembre del 1989 sotto il nome di standard ANSI X3.159-1989. Nel 1990, venne approvato dall'International Organization for Standardization (ISO) con la sigla ISO/IEC 9899:1990. Questa versione del linguaggio è abitualmente indicata come C89 o C90 per distinguere dalla versione originale del C, chiamata K&R C. L'Appendice C riassume le maggiori differenze tra il C89 e il K&R C.

Il linguaggio incontrò alcuni cambiamenti nel 1995 (descritti in un documento conosciuto come Amendment 1). Cambiamenti più significativi avvennero nel 1999 all'atto della pubblicazione del nuovo standard ISO/IEC 9899:1999. Il linguaggio descritto in questo standard è comunemente conosciuto come C99. I termini "ANSI C", "ANSI/ISO C" e "ISO C" – un tempo utilizzati per indicare il C98 – sono attualmente ambigui a causa dell'esistenza di due standard.

C99

Considerato che il C99 non è ancora universalmente diffuso e per la necessità di mantenere milioni (se non miliardi) di righe di codice scritte con la vecchia versione del C, userò una speciale icona (apposta nel margine sinistro) per indicare discussioni su caratteristiche che sono state aggiunte in C99. Un compilatore che non riconosce queste caratteristiche non è "C99-compliant", ovvero non è conforme al nuovo standard ISO. Se la storia davvero insegna, ci vorranno anni affinché tutti i compilatori C divengano conformi al C99, se mai lo diventeranno veramente. L'Appendice B elenca le maggiori differenze tra il C99 e il C89.

Linguaggi basati sul C

Il C ha avuto un'enorme influenza sui linguaggi di programmazione moderni, molti dei quali attingono in maniera considerevole da esso. Dei tanti linguaggi basati sul C, alcuni spiccano sugli altri in modo speciale:

- **C++** include tutte le caratteristiche del C, ma aggiunge le classi ed altre caratteristiche per supportare la programmazione orientata agli oggetti.
- **Java** è basato sul C++ e quindi eredita molte delle caratteristiche del C.
- **C#** è un più recente linguaggio derivato dal C++ e da Java.
- **Perl** era originariamente un linguaggio di scripting piuttosto semplice, con l'andare del tempo è cresciuto e ha adottato molte delle caratteristiche del C.

Considerata la popolarità di questi più moderni linguaggi, è logico chiedersi se valga la pena di imparare il C. Penso che la risposta sia affermativa per diverse ragioni. Primo, imparare il C può portare a una maggiore comprensione delle caratteristiche di C++, Java, C#, Perl e degli altri linguaggi basati sul C. I programmati che imparano per primo uno di questi linguaggi spesso finiscono per non padroneggiare le caratteristiche di base che sono ereditate dal C. Secondo, ci sono molti vecchi programmi C e può capitare di dover leggere e fare manutenzione a quel genere di codice. Terzo, il C è ancora molto usato per sviluppare nuovo software, specialmente in situazioni dove memoria o potenza di calcolo sono limitate oppure dove la semplicità del C è preferibile.

Se non avete ancora utilizzato nessuno dei più recenti linguaggi basati sul C, allora scoprirete che questo libro costituisce un'eccellente preparazione per impararli. Infatti sono enfatizzati l'astrazione dei dati, il cosiddetto *information hiding*, e altri principi che svolgono un largo ruolo nella programmazione a oggetti. Il C++ include tutte le caratteristiche del C, per questo motivo tutto ciò che imparerete in questo libro potrà essere riutilizzato nel caso in cui decidiate di passare al C++ in un momento successivo. Allo stesso modo, molte delle caratteristiche del C possono essere ritrovate in altri linguaggi basati sul C stesso.

1.2 Pregi e debolezze del C

Come qualsiasi altro linguaggio di programmazione anche il C ha i suoi punti di forza e le sue debolezze. Entrambi derivano dall'utilizzo originale del linguaggio (scrivere sistemi operativi ed altri software di sistema) e dalla filosofia su cui si basa.

- **Il C è un linguaggio di basso livello.** Poiché è un linguaggio adatto alla programmazione di sistema, il C fornisce accesso a concetti a livello macchina (byte e indirizzi, per esempio) che altri linguaggi cercano di nascondere. Il C, inoltre, affinché i programmi possano funzionare più velocemente, rende disponibili operazioni che corrispondono strettamente alle istruzioni proprie del computer. Dato che i programmi applicativi si affidano al sistema operativo per l'input/output, la gestione dei file e di numerosi altri servizi, allora quest'ultimo non può permettersi di essere lento.
- **Il C è un "piccolo" linguaggio.** Il C fornisce un numero molto limitato di caratteristiche rispetto a molti linguaggi esistenti. (Il manuale di riferimento della seconda edizione del K&R copre l'intero linguaggio in 49 pagine.) Per mantenere piccolo il numero di funzionalità, il C si appoggia pesantemente su una "libreria" di funzioni standard. (Una "funzione" è simile a quello che in altri linguaggi può essere chiamato "procedura", "subroutine" o "metodo").
- **Il C è un linguaggio permissivo.** Il C assume che si conosca quello che si sta facendo e concede così un maggior grado di libertà rispetto a molti altri linguaggi. Inoltre non è provvisto del controllo dettagliato degli errori presente in altri linguaggi.

Pregi

I punti di forza del C aiutano a spiegare il motivo della popolarità di questo linguaggio.

- **Efficienza.** L'efficienza è stata uno dei vantaggi del C fin dai suoi esordi. Il C era stato pensato per applicazioni dove tradizionalmente veniva utilizzato il linguaggio assembly, era cruciale quindi che i programmi scritti in C potessero girare velocemente e con una quantità di memoria limitata.
- **Portabilità.** Sebbene la portabilità dei programmi non fosse uno degli obiettivi principali del C, essa è divenuta uno dei principali punti di forza del linguaggio. Quando un programma deve poter funzionare su macchine che vanno dai PC fino ai supercalcolatori, spesso è scritto in C. Una delle ragioni della sua portabilità è che – grazie all'associazione con UNIX inizialmente e più tardi con gli standard ANSI/ISO – il linguaggio non si è frammentato in dialetti incompatibili tra loro. I compilatori C, inoltre, sono piccoli e possono essere scritti facilmente, per questo sono largamente diffusi. Infine, il C stesso ha delle caratteristiche per supportare la portabilità (sebbene non si possa fare nulla per evitare che i programmatore scrivano programmi non portabili).
- **Potenza.** La grande collezione di tipi di dato posseduta da C lo rende un linguaggio molto potente. In C è spesso possibile ottenere molto con poche linee di codice.
- **Flessibilità.** Sebbene originariamente il C fosse pensato per la programmazione di sistema, non ha ereditato alcuna restrizione che lo costringa a operare solamente in quel settore. Attualmente viene utilizzato per applicazioni di tutti i tipi, dai sistemi embedded fino ad applicazioni commerciali per l'elaborazione di dati. Il C, inoltre, impone veramente poche restrizioni all'uso delle sue funzionalità; operazioni che

non sarebbero consentite in altri linguaggi, spesso lo sono in C. Per esempio il C ammette la somma di un carattere con un valore intero (oppure con un numero floating point). Questa flessibilità può rendere la programmazione più facile, sebbene possa permettere a diversi bug di insinuarsi nel codice.

- **Libreria Standard.** Uno dei più grandi punti di forza del C è la sua libreria standard, che contiene centinaia di funzioni deputate all'input/output, alla manipolazione delle stringhe, alla gestione della memorizzazione e a molte altre attività utili.
- **Integrazione con UNIX.** Il C è particolarmente potente se combinato con UNIX (inclusa la sua popolare variante conosciuta come Linux). Infatti alcuni dei tool presenti in UNIX presuppongono una conoscenza del C da parte dell'utente.

Debolezze

Le debolezze del C sorgono dalla stessa fonte di molti dei suoi pregi: la sua stretta vicinanza alla macchina. Di seguito vengono elencati alcuni dei più noti problemi riscontrati per questo linguaggio.

- **I programmi C possono essere inclini agli errori.** La flessibilità del C lo rende un linguaggio incline agli errori. Errori di programmazione che sarebbero rilevati in altri linguaggi di programmazione non possono essere individuati dai compilatori C. Sotto questo aspetto, il C è molto simile al linguaggio assembly, dove la maggior parte degli errori non vengono scoperti fino a che il programma non viene messo in funzione. A peggiorare le cose poi, è il fatto che il C contiene numerose trappole per i programmati non accorti. Nei prossimi capitoli vedremo come un segno di punto e virgola in più del dovuto possa causare dei loop infiniti, oppure come la mancanza del simbolo “&” possa causare il crash di un programma.
- **I programmi C possono essere difficili da capire.** Sebbene il C sia, sotto molti punti di vista, un piccolo linguaggio, possiede un certo numero di caratteristiche e funzionalità non presenti in tutti i linguaggi di programmazione (e che di conseguenza molto spesso non vengono capite). Queste funzionalità possono essere combinate in una grande varietà di modi, molti dei quali – sebbene ovvi all'autore originale del programma – possono risultare difficili da capire per gli altri programmati. Un altro problema è la natura succinta e stringata dei programmi. Il C è stato progettato quando l'interazione con il computer era estremamente tediosa; di conseguenza il linguaggio venne mantenuto conciso di proposito, al fine di minimizzare il tempo richiesto all'immissione e alla scrittura dei programmi. La flessibilità del C può essere inoltre un fattore negativo, i programmati che sono troppo esperti e capaci possono, per loro interesse personale, scrivere programmi pressoché impossibili da comprendere.
- **I programmi C possono essere difficili da modificare.** Lunghi programmi scritti in C possono essere particolarmente difficili da modificare, se non sono stati sviluppati tenendo presente la necessità di manutenzione del codice. I moderni linguaggi di programmazione dispongono di funzionalità come le classi e i package che supportano la suddivisione dei programmi lunghi in sezioni di codice molto più gestibili. Il C sfortunatamente sente la mancanza di queste caratteristiche.

Il C offuscato

Anche i suoi più ardenti ammiratori ammettono che il C può essere difficile da leggere. L'annuale competizione internazionale del codice C offuscato (International Obfuscated C Code Contest) attualmente incoraggia i partecipanti a scrivere programmi il più possibile confusi. I vincitori sono veramente sconcertanti, ne è esempio il "Best Small Program" del 1990:

```
v,i,j,k,l,s,a[99];
main()
{
    for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,j+=(v=j<s&&
(!k&&!printf(2+"\n\n%c"-(!l<<!j)," #Q"[1^v?(l^j)&1:2])&&
++l||a[i]<s&&v>i+j&&v+i-j))&&!(l=s),v||(i==j?a[i+=k]=0:
++a[i])>=s*k&&++a[-i])
    ;
}
}
```

Questo programma, scritto da Doron Osovianski e Baruch Nissenbaum, stampa tutte le soluzioni del problema chiamato Eight Queens (il problema di disporre otto regine su una scacchiera in modo che nessuna regina attacchi nessun'altra). Infatti, il programma funziona per un qualsiasi numero di regine compreso tra quattro e 99. Per vedere altri programmi vincitori della competizione visitate il sito internet www.ioccc.org.

Utilizzo efficace del C

Utilizzare il C efficacemente significa sfruttare i vantaggi dei suoi punti di forza evitando contemporaneamente le sue debolezze. Qui sono elencati alcuni suggerimenti.

- **Imparare ad evitare le trappole del C.** Suggerimenti per evitare le trappole sono sparsi in tutto il libro – basta cercare il simbolo Δ . Per una lista più estesa si fa riferimento al volume di Andrews Koenig *C Traps and Pitfalls* (Addison-Wesley 1989). I moderni compilatori sono in grado di rilevare le trappole più comuni e lanciare dei warning, tuttavia nessun compilatore è in grado di trovare tutte le insidie presenti all'interno del codice.
- **Utilizzare tool software per rendere i programmi più affidabili.** I programmatore C sono sviluppatori (e utilizzatori) di tool molto prolifici. Uno dei più famosi strumenti per il C è chiamato *lint*. *lint*, che tradizionalmente viene fornito con UNIX, può sottoporre un programma ad analisi molto più intensive per quel che riguarda gli errori, rispetto alla maggior parte dei compilatori. Se *lint* (oppure un programma simile) è disponibile, allora è una buona idea utilizzarlo. Un altro strumento utile è il debugger. A causa della natura del C, molti dei bachi di un programma non possono essere rilevati da un compilatore; questi bachi infatti si manifestano sotto forma di errori di run-time oppure output incorretto. Di conseguenza, utilizzare un buon debugger è d'obbligo per i programmatore C.
- **Trarre vantaggio da librerie di codice esistente.** Uno dei benefici dell'utilizzare il C è che anche molte altre persone lo utilizzano; così c'è una buona

probabilità che qualcuno abbia già scritto del codice che potremmo impiegare nei propri programmi. Il codice C è spesso accorpato in librerie (collezioni di funzioni); quindi impiegare una libreria adatta allo scopo è un buon metodo per ridurre gli errori e risparmiarsi uno sforzo considerevole durante la programmazione. Librerie per i compiti più comuni, incluso lo sviluppo di interfacce utente, grafica, comunicazioni, utilizzo del database e networking sono immediatamente disponibili. Alcune librerie sono di pubblico dominio, alcune sono open source ed alcune sono in vendita.

- **Adottare un insieme consistente di convenzioni nel codice.** Una convenzione nella scrittura del codice è una regola stilistica che un programmatore decide di adottare anche se non è richiesta dal linguaggio. Le buone convenzioni aiutano a rendere i programmi più uniformi, più facili da leggere e da modificare. Il loro utilizzo è importante con qualsiasi linguaggio di programmazione, ma con il C lo è in modo particolare. Come è già stato detto, la natura altamente flessibile del C permette ai programmatore di scrivere codice totalmente illeggibile. Gli esempi di programmazione contenuti in questo libro seguono un dato insieme di convenzioni, tuttavia ci sono altre convenzioni ugualmente valide (di tanto in tanto discuteremo di alcune alternative). Qualunque sia il set di convenzioni che decidiate di utilizzare è meno importante della necessità di adottare delle convenzioni e di seguirle fedelmente.
- **Evitare "trucchetti" e codice eccessivamente complesso.** Il C incoraggia una programmazione fatta di espedienti. Ci sono diversi modi per ottenere il medesimo risultato e i programmatore sono spesso tentati di scegliere il metodo più conciso. Non lasciatevi tentare: la soluzione più breve è spesso quella di più difficile comprensione. In questo libro illustrerò uno stile che è ragionevolmente conciso ma, nonostante ciò, semplice e chiaro.
- **Attenersi allo standard.** Molti compilatori C forniscono caratteristiche e librerie che non sono parte degli standard C89 o C99. Per ragioni di portabilità è preferibile evitare l'utilizzo di *feature* e librerie non standard a meno che non risultino strettamente necessarie.

Domande & Risposte

D: Cos'è la sezione D&R?

R: Lieto che lo abbiate chiesto. La sezione D&R (domande e risposte) che appare alla fine di ogni capitolo si prefigge molteplici scopi. Il fine principale è affrontare domande che vengono poste frequentemente da chi studia il C. I lettori possono partecipare (più o meno) a un dialogo con l'autore, quasi come se stessero frequentando una delle mie lezioni. Un altro scopo di D&R è fornire informazioni aggiuntive sugli argomenti coperti all'interno del capitolo. Alcuni avranno già avuto esperienze di programmazione con altri linguaggi, mentre altri si avvicineranno alla programmazione per la prima volta. I lettori con esperienza in una varietà di linguaggi potranno essere soddisfatti da una breve spiegazione e da un paio di esempi, mentre ai lettori con meno esperienza potrebbe essere necessaria qualche spiegazione in più. Riassumendo: se riterrete la copertura di un certo argomento troppo concisa, allora

controllate la sezione D&R per maggiori dettagli. Occasionalmente la sezione D&R discuterà delle differenze più comuni tra i compilatori C. Per esempio, parleremo di alcune caratteristiche di particolari compilatori, che vengono frequentemente impiegate sebbene non aderiscano allo standard.

D: Cosa fa esattamente lint? [p.6]

R: lint controlla un programma C rispetto a una serie di potenziali errori, inclusi – ma non solo questi – la sospetta combinazione di tipi, la presenza di variabili inutilizzate, di codice non raggiungibile e di codice non portabile. lint produce un elenco di messaggi di diagnostica che devono essere vagliati dal programmatore. Il vantaggio nell'utilizzare lint è che permette di individuare errori che sfuggono al compilatore. Un altro problema è dato dal fatto che lint produce anche centinaia di messaggi, ma solamente una frazione di questi si riferisce di fatto a veri errori.

D: Da dove deriva il nome lint?

R: A differenza di molti altri tool di UNIX, lint non è un acronimo. Il suo nome deriva dal modo in cui estrae pezzi di "lanugGINE" da un programma.

D: Come posso ottenere una copia di lint?

R: lint è un'utility standard di UNIX. Fortunatamente sono disponibili versioni di lint fornite da terze parti. Una versione denominata splint (Secure Programming Lint) è inclusa in molte distribuzioni Linux e può essere scaricata gratuitamente da www.splint.org.

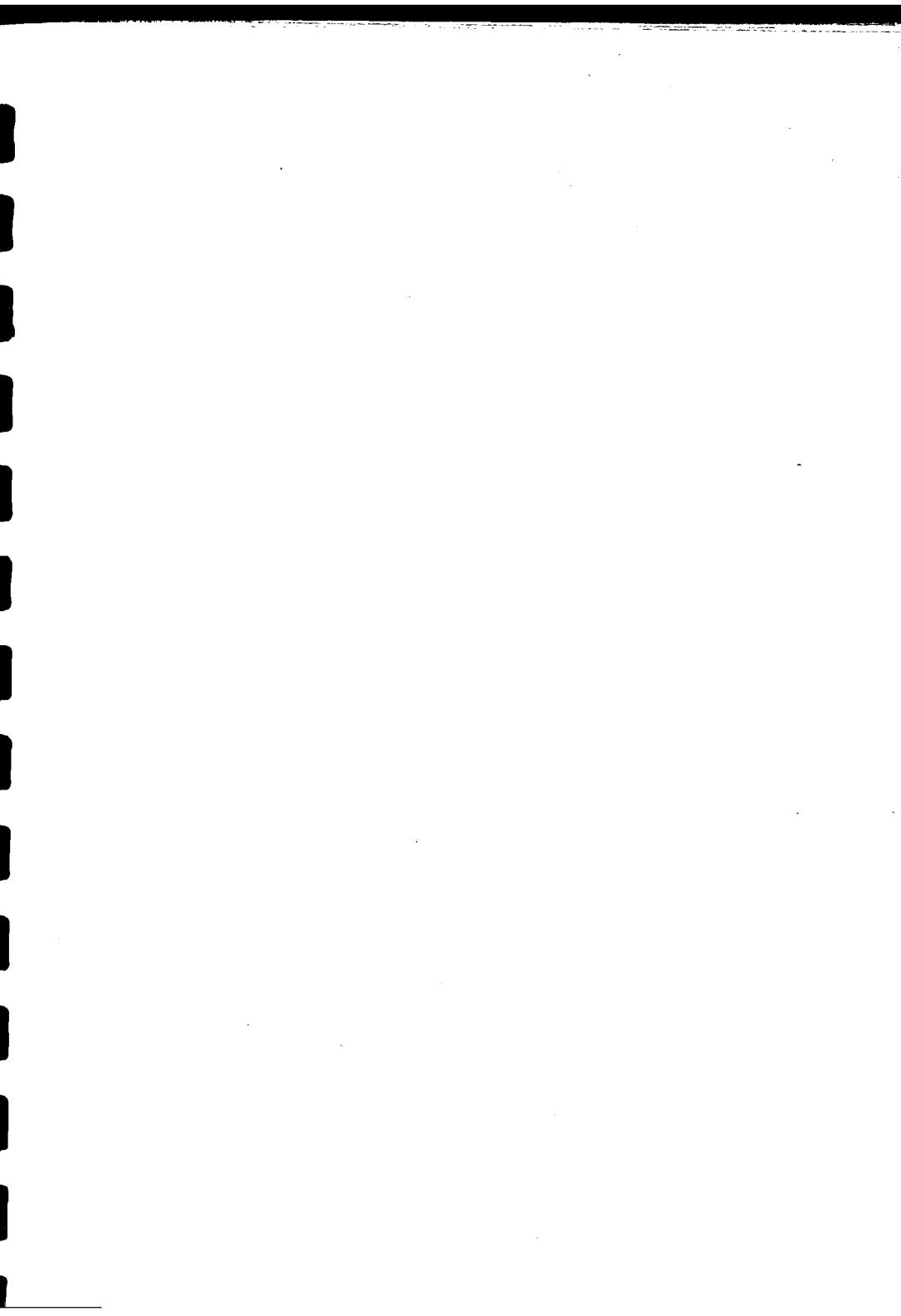
D: È possibile forzare un compilatore a fare un lavoro più accurato di controllo degli errori senza dover usare lint?

R: Sì. Molti compilatori faranno un lavoro più accurato di controllo se viene loro richiesto. Oltre al controllo degli errori (ovvero di indiscusse violazioni delle regole del C), molti compilatori producono anche messaggi di avvertimento che indicano punti potenzialmente problematici. Alcuni compilatori hanno più di un "livello di warning"; selezionando un livello più alto il compilatore controllerà un numero maggiore di problemi rispetto alla scelta di un livello più basso. Se il vostro compilatore supporta diversi livelli di warning, è buona norma selezionare il livello più alto obbligando il compilatore a eseguire il controllo più accurato che è in grado di effettuare. Le opzioni di controllo degli errori per il compilatore GCC [[GCC > 2.1](#)], che è distribuito con Linux, sono discussi nella sezione D&R alla fine del Capitolo 2.

D: Vorrei rendere i miei programmi il più possibile affidabili. Sono disponibili altri tool oltre a lint e ai debugger?

* Le domande indicate con l'asterisco riguardano materiale avanzato e spesso si riferiscono ad argomenti che vengono coperti nei capitoli successivi. I lettori con una buona esperienza in programmazione, possono affrontare immediatamente queste domande.

R: Sì. Altri strumenti molto comuni includono i "bound-checker" e i "leak-finder". Il C non richiede che vengano controllati i limiti di un array; un bound-checker aggiunge questa funzionalità. Un leak-finder aiuta a trovare i "memory leak": ovvero i blocchi di memoria allocati dinamicamente e che non vengono mai deallocati.



2 Fondamenti di C

Questo capitolo introduce diversi concetti base, che includono: le direttive del pre-processore, le funzioni, le variabili e le istruzioni di cui avremo bisogno per scrivere anche il più semplice dei programmi. I capitoli successivi tratteranno questi argomenti con maggiore dettaglio.

Per iniziare, la Sezione 2.1 presenta un piccolo programma C e spiega come compilarlo ed eseguirne il *linking*. La Sezione 2.2 discute su come generalizzare il programma, mentre la Sezione 2.3 mostra come aggiungere delle note esplicative conosciute come commenti. La Sezione 2.4 invece introduce le variabili che memorizzano dei dati che possono cambiare durante l'esecuzione di un programma. La Sezione 2.5 illustra l'utilizzo della funzione `scanf` per leggere i dati e inserirli nelle variabili. Come vedremo nella Sezione 2.6, anche alle costanti – dati che non cambieranno durante l'esecuzione del programma – può essere dato un nome. Infine la Sezione 2.7 illustra le regole del C per la scelta dei nomi degli identificatori, mentre la Sezione 2.8 fornisce delle regole generali per la stesura di un programma.

2.1 Scrivere un semplice programma

I programmi C, in contrasto rispetto a quelli scritti in altri linguaggi, richiedono pochissimo codice di contorno – un programma completo può anche essere di poche righe.

PROGRAMMA

Visualizzare il *bad pun*

Il primo programma che troviamo nel libro di Kernighan e Ritchie, il classico *The C Programming Language*, è estremamente corto e non fa nient'altro che scrivere il messaggio “hello, world”. A differenza di altri autori C, non utilizzerò questo programma come primo esempio. Sosterrò invece un'altra tradizione del C: il *bad pun*. Questo è il *bad pun*:

To C, or not to C: that is the question.

Il seguente programma, che chiameremo `pun.c`, visualizza il messaggio ogni volta che viene eseguito.

```
pun.c #include <stdio.h>
int main(void)
{
    printf("To C, or not to C: that is the question.\n");
    return 0;
}
```

Nella Sezione 2.2 la struttura del programma viene spiegata con un certo dettaglio. Per ora farò solamente qualche breve osservazione. La linea

```
#include <stdio.h>
```

è necessaria per "includere" le informazioni riguardanti la libreria standard di I/O (input/output) del C. Il codice eseguibile si trova all'interno della sezione `main` che rappresenta la parte principale del programma. L'unica istruzione all'interno del `main` è il comando per stampare il messaggio desiderato, infatti la `printf` è la funzione della libreria standard di I/O adatta a produrre dell'output opportunamente formattato. Il codice `\n` serve per avvertire la `printf` di avanzare alla linea successiva, dopo la stampa del messaggio. L'istruzione

```
return 0;
```

indica che il programma, quando termina, "restituisce" il valore 0 al sistema operativo.

Compilazione e linking

A dispetto della sua brevità, eseguire `pun.c` è più complicato di quello che potreste aspettarvi. Per prima cosa dobbiamo creare il file chiamato `pun.c` contenente il programma (un qualsiasi *editor* di testo andrà bene). Il nome del file non ha importanza tuttavia l'estensione `.c` è un requisito per molti compilatori.

Successivamente dobbiamo convertire il programma in una forma che il computer possa eseguire. Per un programma C questo coinvolge tre passi.

- **Preprocessamento.** Il programma viene prima dato in pasto a un **preprocessore**, il quale obbedisce ai comandi che iniziano con `#` (conosciuti come **direttive**). Un preprocessore è simile a un editor, può aggiungere parti al programma e introdurre delle modifiche.
- **Compilazione.** Il programma modificato deve andare a un compilatore, il quale lo traduce in istruzioni macchina (*object code*). Nonostante questo il programma non è ancora del tutto pronto per essere eseguito.
- **Linking.** Nel passo finale, il *linker* combina il codice oggetto prodotto dal compilatore con del codice addizionale, necessario per rendere il programma completamente eseguibile.

Fortunatamente questo processo molto spesso viene automatizzato, per questo motivo non lo troverete eccessivamente gravoso. Infatti il preprocessore solitamente è integrato con il compilatore e quindi molto probabilmente non lo noterete nemmeno lavorare.

I comandi necessari per compilare e per il *linking* variano, essi dipendono sia dal compilatore che dal sistema operativo. Negli ambienti UNIX, il compilatore C so-

litamente si chiama cc. Per compilare e fare il linking del programma pun.c si deve immettere il seguente comando in un terminale o in una finestra a riga di comando:

```
% cc pun.c
```

(Il carattere % è il *prompt* UNIX, non è qualcosa che dovete scrivere.) Il linking è automatico quando si utilizza cc, non è necessario nessun comando aggiuntivo.

Dopo aver compilato ed eseguito il linking, per default cc rilascia il programma eseguibile in un file chiamato a.out. Il linker cc ha molte opzioni, una di queste (l'opzione -o) ci permette di scegliere il nome del file contenente il programma eseguibile. Per esempio, se vogliamo una versione eseguibile del programma pun.c chiamata pun, allora immetteremo il comando seguente:

```
% cc -o pun pun.c
```

Il compilatore GCC

Uno dei più popolari compilatori C è GCC, che viene fornito con Linux ma è disponibile anche per altre piattaforme. L'utilizzo di questo compilatore è simile al tradizionale compilatore UNIX cc. Per esempio, per compilare il programma pun.c useremo il seguente comando:

```
% gcc -o pun pun.c
```

 La sezione D&R alla fine di questo capitolo fornisce molte informazioni a riguardo di GCC.

Sistemi di sviluppo integrati

Fino a qui è stato assunto l'uso di un compilatore a "riga di comando", ovvero che viene invocato immettendo un comando in una speciale finestra fornita dal sistema operativo. L'alternativa è l'utilizzo di un sistema di sviluppo integrato (IDE, *Integrated Development Environment*): un pacchetto software che ci permette di scrivere, compilare, fare il linking, eseguire e persino fare il debug di un programma senza mai lasciare l'ambiente di sviluppo. I componenti di un IDE sono progettati per lavorare assieme. Per esempio, quando un compilatore rileva un errore in un programma, può far sì che l'editor sottolinei la linea che contiene l'errore. Ci sono grandi differenze tra i diversi IDE, per questo motivo non ne parlerò più all'interno di questo libro. In ogni caso, vi raccomanderei di controllare quali sono gli IDE disponibili per la vostra piattaforma.

2.2 La struttura generale di un programma

Diamo un'occhiata più da vicino a pun.c e vediamo come poterlo generalizzare. I programmi C più semplici hanno la forma

```
direttive  
int main(void)  
{  
    istruzioni  
}
```

In questo modello, e in modelli analoghi presenti in altre parti del libro, gli oggetti scritti con carattere Courier appariranno in un programma C esattamente come sono, mentre gli oggetti scritti in corsivo rappresentano del testo al quale deve provvedere il programmatore.

DAR

Fate caso a come le parentesi graffe indichino l'inizio e la fine del `main`. Il C utilizza le parentesi { e } praticamente allo stesso modo in cui altri linguaggi utilizzano parole come `begin` ed `end`. Quanto appena detto illustra uno dei punti generali riguardo al C, ovvero che il linguaggio si affida ad abbreviazioni e a simboli speciali. Questa è una delle ragioni per cui i programmi sono così concisi (o criptici, per dirla in modo meno cortese).

Anche il più semplice programma C si basa su tre componenti chiave del linguaggio: le direttive (comandi di editing che modificano il programma prima della compilazione), le funzioni (blocki di codice eseguibile cui viene dato un nome, il `main` ne è un esempio) e le istruzioni (comandi che devono essere eseguiti quando il programma è in funzione). Vediamo ora queste componenti in maggiore dettaglio.

Direttive

Prima che un programma C venga compilato deve essere modificato da un preprocessore, i comandi indirizzati a quest'ultimo vengono chiamati direttive. I Capitoli 14 e 15 discutono delle direttive in dettaglio. Per adesso, siamo interessati solo alle direttive `#include`.

Il programma `pun.c` inizia con la linea
`#include <stdio.h>`

Questa direttiva indica che le informazioni contenute in `<stdio.h>` devono essere "inclusse" nel programma prima che venga compilato. L'header `<stdio.h>` contiene informazioni riguardanti la libreria standard di I/O del C. Il C possiede un certo numero di header [header > 15.2], come `<stdio.h>`, ognuno dei quali contiene informazioni riguardanti una porzione della libreria standard. La ragione per la quale stiamo includendo `<stdio.h>` è che il C, a differenza di altri linguaggi di programmazione, non ha dei comandi incorporati di "lettura" e "scrittura". La possibilità di eseguire dell'input e dell'output è fornita invece dalle funzioni presenti nella libreria standard.

Le direttive iniziano sempre con il carattere # che le distingue dagli altri oggetti presenti in un programma C. Per default le direttive sono lunghe una sola riga e non vi è nessun punto e virgola o qualche altro indicatore speciale alla loro fine.

Funzioni

Le funzioni sono come le "procedure" o le *subroutine* in altri linguaggi di programmazione ovvero blocchi per mezzo dei quali i programmi vengono costruiti, infatti un programma C non è molto di più di una collezione di funzioni. Le funzioni ricadono in due categorie: quelle scritte dal programmatore e quelle fornite come parte dell'implementazione del C. Mi riferirò a queste ultime come alle funzioni di libreria (*library functions*), in quanto appartengono a una "libreria" di funzioni che sono fornite assieme al compilatore.

Il termine "funzione" deriva dalla matematica dove una funzione è una regola per calcolare un valore a partire da uno o più argomenti dati:

$$f(x) = x + 1$$

$$g(x, y) = y^2 - z^2$$

Il C invece utilizza il termine "funzione" in modo meno restrittivo. In C una funzione è semplicemente un raggruppamento di una serie di istruzioni al quale è stato assegnato un nome. Alcune funzioni calcolano un valore, altre no. Una funzione che calcola un valore utilizza l'istruzione `return` per specificare il valore che deve restituire. Per esempio, una funzione che somma 1 al suo argomento dovrà eseguire l'istruzione:

```
return x+1;
```

mentre una funzione che calcola la differenza dei quadrati dei suoi argomenti deve eseguire l'istruzione

```
return y*y - z*z;
```

Sebbene un programma C possa essere composto da molte funzioni, solo la funzione `main` è obbligatoria. La funzione `main` è speciale: viene invocata automaticamente quando il programma viene eseguito. Fino al Capitolo 9, dove impareremo come scrivere altre funzioni, il `main` sarà l'unica funzione dei nostri programmi.



Il nome `main` è obbligatorio, non può essere sostituito con `begin` o `start` oppure `MAIN`

Se il `main` è una funzione, questa restituisce un valore? Sì, restituisce un codice di stato che viene passato al sistema operativo quando il programma termina. Diamo un'altra occhiata al programma `pun.c`:

```
#include <stdio.h>

int main(void)
{
    printf("To C, or not to C: that is the question.\n");
    return 0;
}
```

La parola `int`, che si trova immediatamente prima della parola `main`, indica che la funzione `main` restituisce un valore intero, mentre la parola `void` all'interno delle parentesi tonde indica che `main` non ha argomenti.

L'istruzione

```
return 0;
```

ha due effetti: causa la fine della funzione `main` (e quindi la fine del programma) e indica che `main` restituisce il valore 0. Discuteremo del valore restituito da `main` in un capitolo successivo [valore restituito dal `main` > 9.5]. Per ora la funzione `main` ritornerà sempre il valore 0 indicando così che il programma è terminato normalmente.

Il programma termina ugualmente anche se non c'è nessuna istruzione `return` alla fine del `main`, tuttavia in quel caso molti compilatori produrranno un messaggio di

warning (perché si suppone che la funzione ritorni un valore intero quando invece non lo fa).

Istruzioni

Un'istruzione è un comando che viene eseguito quando il programma è in funzione. Esploreremo le istruzioni più avanti nel libro, principalmente nei Capitoli 5 e 6. Il programma pun.c utilizza solamente due tipi di istruzioni. Una è l'istruzione return, l'altra è la **chiamata a funzione** (*function call*). Chiedere ad una funzione di compiere la sua mansione viene detto **chiamare** la funzione. Il programma pun.c, ad esempio, chiama la funzione printf per visualizzare una stringa sullo schermo:

```
printf("To C, or not to C: that is the question.\n");
```

Il C ha bisogno che ogni istruzione termini con un punto e virgola (e, come ogni buona regola, anche quella appena citata ha un'eccezione: l'istruzione composta che incontreremo più avanti [**istruzione composta > 5.2**]). Infatti il punto e virgola serve per indicare al compilatore dove termina l'istruzione visto che questa potrebbe svilupparsi su più righe e non sempre è facile identificarne la fine. Al contrario, le direttive di norma sono lunghe una sola riga e *non* terminano con un punto e virgola.

Stampare le stringhe

La printf è una potente funzione che esamineremo nel Capitolo 3. Fin qui abbiamo utilizzato printf solo per stampare una **stringa testuale** – cioè una serie di caratteri racchiusi tra doppi apici. Quando la printf stampa una stringa testuale non visualizza i doppi apici.

La funzione printf non avanza automaticamente alla linea successiva dell'output quando termina la stampa. Per indicare alla printf di avanzare di una linea dobbiamo aggiungere \n (il carattere **new-line**) alla stringa che deve essere stampata.

Scrivere il carattere new-line fa terminare la linea corrente e per conseguenza l'output finisce sulla linea successiva. Per illustrare questo concetto consideriamo l'effetto di rimpiazzare l'istruzione

```
printf("To C, or not to C: that is the question.\n");
```

con due chiamate alla printf:

```
printf("To C, or not to C: ");
printf("that is the question.\n");
```

La prima chiamata scrive To C, or not to C:. La seconda chiamata scrive that is the question. e avanza sulla riga successiva. L'effetto complessivo è lo stesso della printf originale – l'utente non potrà notare la differenza.

Il carattere new-line può apparire anche più volte in una stringa testuale, per visualizzare il messaggio

Brevity is the soul of wit.

--Shakespeare

possiamo scrivere

```
printf("Brevity is the soul of wit.\n --Shakespeare\n");
```

2.3 Commenti

Al nostro programma `pun.c` manca qualcosa di importante: la documentazione. Ogni programma dovrebbe contenere delle informazioni identificative: il nome del programma, la data di scrittura, l'autore, lo scopo del programma e così via. In C queste informazioni vengono messe all'interno dei **commenti**. Il simbolo `/*` indica l'inizio di un commento e il simbolo `*/` ne indica la fine:

```
/* Questo è un commento */
```

I commenti possono apparire praticamente ovunque in un programma, sia su righe separate che sulla medesima riga sulla quale si trova altro testo appartenente al programma. Ecco come potrebbe apparire `pun.c` con l'aggiunta di alcuni commenti all'inizio:

```
/* Nome: pun.c */
/* Scopo: stampare il bad pun. */
/* Autore: K. N. King */

#include <stdio.h>

int main(void)
{
    printf("To C, or not to C: that is the question.\n");
    return 0;
}
```

I commenti possono anche estendersi su più righe. Una volta che vede il simbolo `/*` il compilatore legge (e ignora) qualsiasi cosa lo segua fino a che non incontra il simbolo `*/`. Se lo preferiamo, possiamo combinare una serie di brevi commenti all'interno di un commento lungo:

```
/* Nome: pun.c
   Scopo: stampare il bad pun.
   Autore: K. N. King */
```

Un commento come questo può essere difficile da leggere, perché non è facile capire dove sia il suo termine. Mettere `/*` su una riga a sé stante invece ne agevola la lettura:

```
/* Nome: pun.c
   Scopo: stampare il bad pun.
   Autore: K. N. King
*/
```

Possiamo fare ancora di meglio formando una "scatola" attorno al commento in modo da evidenziarlo:

```
*****  
* Nome: pun.c *  
* Scopo: stampare il bad pun. *  
* Autore: K. N. King *  
******/
```

I programmati spesso semplificano i commenti inscatolati omettendo tre dei lati:

```
/*
 * Nome: pun.c
 * Scopo: stampare il bad pun.
 * Autore: K. N. King
 */
```

Un commento breve può venir messo sulla stessa riga di una porzione del programma:

```
int main(void) /* Inizio del main del programma */
```

Un commento come questo viene chiamato a volte "commento a latere" o "winged comment".



Dimenticare di chiudere un commento può far sì che il compilatore ignori parte del vostro programma. Considerate l'esempio seguente:

```
printf("My "); /* dimenticato di chiudere un commento...
printf("cat ");
printf("has "); /* quindi finisce qui */
printf("fleas ");
```

Aver dimenticato di chiudere il primo commento fa sì che il compilatore ignori le due istruzioni intermedie e che l'esempio stampi a video My fleas.



Il C99 prevede un secondo tipo di commenti, i quali iniziano con // (due barre adiacenti):

```
//Questo è un commento
```

Questo stile di commento termina automaticamente alla fine di una riga. Per creare un commento più lungo di una riga possiamo utilizzare o il vecchio stile /* - */ oppure mettere // all'inizio di ogni riga:

```
// Nome: pun.c
// Scopo: stampare il bad pun.
// Autore: K. N. King
```

Il nuovo stile per i commenti ha un paio di vantaggi importanti. Primo: il fatto che il commento termini automaticamente alla fine di ogni riga esclude il pericolo che un commento non terminato causi l'esclusione accidentale di una parte del programma. Secondo: i commenti su più righe risaltano meglio grazie al // che è richiesto all'inizio di ogni riga.

2.4 Variabili e assegnamenti

Pochi programmi sono semplici come quello della Sezione 2.1. Molti programmi devono eseguire una serie di calcoli prima di produrre l'output, e quindi necessitano di un modo per memorizzare temporaneamente i dati durante l'esecuzione del

programma. In C, come nella maggior parte dei linguaggi di programmazione, questi luoghi di memorizzazione vengono chiamati **variabili**.

Tipi

Ogni variabile deve avere un **tipo** che specifichi la tipologia di dati che dovrà contenere. Il C ha un'ampia varietà di tipi, ma per ora ci limiteremo a usarne solamente due: **int** e **float**. È particolarmente importante scegliere il tipo appropriato: da esso dipende il modo in cui la variabile viene memorizzata e le operazioni che si possono compiere su essa. Il tipo di una variabile numerica determina il numero più grande e quello più piccolo che la variabile stessa può contenere, determina inoltre se delle cifre decimali sono ammesse o meno.

Una variabile di tipo **int** (abbreviazione di *integer*) può memorizzare un numero intero come 0, 1, 392 oppure -2553. Tuttavia l'intervallo dei possibili valori è limitato [**intervallo dei valori degli int > 7.1**]: il più grande valore per un **int** è tipicamente 2.147.483.647 ma potrebbe essere anche più piccolo, come 32.767.



Una variabile di tipo **float** (abbreviazione di *floating-point*) può memorizzare numeri più grandi rispetto a una variabile di tipo **int**, inoltre una variabile **float** può contenere numeri con cifre dopo la virgola, come 379,125. Le variabili **float**, però, hanno delle controindicazioni, infatti i calcoli aritmetici su questo tipo di variabili possono essere più lenti rispetto a quelli sui numeri di tipo **int**. Inoltre la cosa più importante da tener presente è che spesso il valore di una variabile **float** è solamente un'approssimazione del numero che è stato memorizzato in essa. Se memorizziamo il valore 0,1 in una variabile **float**, potremmo scoprire più tardi che la variabile contiene il valore 0,09999999999999987 a causa dell'errore di arrotondamento.

Dichiarazioni

Le variabili devono essere **dichiarate** – cioè descritte a beneficio del compilatore – prima di poter essere utilizzate. Per dichiarare una variabile dobbiamo prima di tutto specificare il *tipo* della variabile e successivamente il suo *nome* (i nomi delle variabili vengono scelti dal programmatore e sono soggetti alle regole descritte nella Sezione 2.7). Per esempio possiamo dichiarare le variabili **height** e **profit** come segue:

```
int height;
float profit;
```

La prima dichiarazione afferma che **height** è una variabile di tipo **int**, indicando in questo modo che può memorizzare un numero intero. La seconda dichiarazione dice che **profit** è una variabile di tipo **float**.

Se diverse variabili sono dello stesso tipo, le loro dichiarazioni possono essere combinate:

```
int height, length, width, volume;
float profit, loss;
```

Tenete presente che per il compilatore una dichiarazione completa termina con un punto e virgola.

Il nostro primo modello per la funzione `main` non includeva dichiarazioni. Quando il `main` contiene dichiarazioni, queste devono precedere le istruzioni:

```
int main(void)
{
    dichiarazioni
    istruzioni
}
```

Nel Capitolo 9 questo è vero in generale per le funzioni, così come per i blocchi (istruzioni che contengono delle dichiarazioni incorporate al loro interno [blocchi > 10.3]). Per questioni di stile è una buona pratica lasciare una riga vuota tra le dichiarazioni e le istruzioni.

C99

Nel C99 non è necessario che le dichiarazioni vengano messe prima delle istruzioni. Per esempio, il `main` può contenere una dichiarazione, poi un'istruzione, e poi un'altra dichiarazione. Per questioni di compatibilità con vecchi compilatori, i programmi di questo libro non si avvarranno di questa regola. Tuttavia nei programmi C++ e Java è comune non dichiarare le variabili fino a quando non vengono utilizzate per la prima volta, quindi ci si può aspettare che questa pratica diventi popolare anche nei programmi C99.

Assegnamenti

Si può conferire un valore ad una variabile tramite un **assegnamento**. Per esempio, le istruzioni

```
height = 8;
length = 12;
width = 10;
```

assegnano dei valori a `height`, `length` e `width`. I numeri 8, 12 e 10 sono chiamati **costanti**.

Prima che a una variabile possa essere assegnato un valore – o possa essere utilizzata in qualsiasi altra maniera – questa deve essere prima dichiarata. Quindi potremmo scrivere

```
int height;
height = 8;

ma non
height = 8;      /** SBAGLIATO ***
int height;
```

Di solito una costante che viene assegnata ad una variabile di tipo `float` contiene il separatore decimale. Per esempio, se `profit` è una variabile `float`, potremmo scrivere

```
profit = 2150.48;
```

D&R

Dopo ogni costante che contiene il separatore decimale, sarebbe bene aggiungere una lettera `f` (che sta per `float`) se questa viene assegnata ad una variabile di tipo `float`:

```
profit = 2150.48f;
```

Non includere la f potrebbe causare un messaggio di warning da parte del compilatore.

Normalmente a una variabile di tipo int viene assegnato un valore di tipo int, così come a una variabile di tipo float viene assegnato un valore di tipo float. Come vedremo nella Sezione 4.2, mischiare i tipi (come assegnare un valore int a una variabile float, o assegnare un valore float a una variabile int) è possibile sebbene non sia sempre sicuro.

Una volta che a una variabile è stato assegnato un valore, questo può essere utilizzato per calcolare il valore di un'altra variabile:

```
height = 8;
length = 12;
width = 10;
volume = height * length * width; /* volume adesso è uguale a 960 */
```

In C, * rappresenta l'operatore di moltiplicazione. Questa istruzione moltiplica il valore contenuto in height, length e width e assegna il risultato alla variabile volume. In generale il lato destro di un assegnamento può essere una qualsiasi formula (o espressione, nella terminologia C) che includa costanti, variabili e operatori.

Stampare il valore di una variabile

Possiamo utilizzare printf per stampare il valore corrente di una variabile. Per esempio per scrivere il messaggio

Height: h

dove h è il valore corrente della variabile height, useremo la seguente chiamata alla printf:

```
printf("Height: %d\n", height);
```

%d è un segnaposto che indica dove deve essere inserito durante la stampa il valore di height. Osservate la disposizione del \n subito dopo il %d, in modo tale che la printf avanzi alla prossima riga dopo la stampa del valore di height.

Il %d funziona solo per le variabili int, per stampare una variabile float useremo %f al suo posto. Per default %f stampa a video un numero con 6 cifre decimali. Per forzare %f a stampare p cifre dopo la virgola possiamo mettere .p tra il % e la f. Per esempio per stampare la riga

Profit: \$2150.48

Chiameremo la printf in questo modo:

```
printf("Profit: %.2f\n", profit);
```

Non c'è limite al numero di variabili che possono essere stampate da una singola chiamata della printf. Per stampare i valori di entrambe le variabili height e length possiamo usare la seguente chiamata a printf:

```
printf("Heighth: %d Length: %d\n", height, length);
```

PRIMA

Calcolare il peso volumetrico di un pacco

Le compagnie di spedizione non amano particolarmente i pacchi che sono larghi ma molto leggeri perché occupano uno spazio considerevole all'interno di un camion o di un aeroplano. Infatti, capita spesso che le compagnie applichino rincari extra per questo tipo di pacchi basando il costo della spedizione sul loro volume invece che sul loro peso. Negli Stati Uniti il metodo usuale è quello di dividere il volume per 166 (il numero di pollici quadrati ammissibile per una libbra). Se questo numero – il peso "dimensionale" o "volumetrico" – eccede il peso reale del pacco allora il costo della spedizione viene basato sul peso dimensionale (166 è il dividendo per le spedizioni internazionali, il peso dimensionale per una spedizione nazionale invece viene calcolato utilizzando 194).

Ipotizziamo che siate stati assunti da una compagnia di spedizione per scrivere un programma che calcoli il peso dimensionale di un pacco. Dato che siete nuovi al C, deciderete di iniziare scrivendo un programma che calcoli il peso dimensionale di un particolare pacco che ha le dimensioni di 12 pollici \times 10 pollici \times 8 pollici. La divisione viene rappresentata in C con il simbolo $/$, e quindi il modo ovvio per calcolare il peso dimensionale sarebbe:

```
weight = volume / 166;
```

dove weight e volume sono le variabili intere che rappresentano il peso ed il volume del pacco. Sfortunatamente questa formula non è quello di cui abbiamo bisogno. Nel C quando un intero viene diviso per un altro intero il risultato viene "troncato": tutte le cifre decimali vengono perse. Il volume di un pacco di 12 pollici \times 10 pollici \times 8 pollici è di 960 pollici cubici. Dividendo per 166 si ottiene come risultato 5 invece che 5.783, in questo modo abbiamo di fatto arrotondato alla libra inferiore, la compagnia di spedizione invece si aspetta che noi arrotondiamo per eccesso. Una soluzione consiste nel sommare 165 al volume prima di dividerlo per 166:

```
weight = (volume + 165) / 166;
```

Un volume di 166 restituirebbe un peso di 331/166, cioè 1, mentre un volume di 167 restituirebbe 332/166, ovvero 2. Calcolare il peso in questo modo ci dà il seguente programma:

```
#include <stdio.h>

int main(void)
{
    int height, length, width, volume, weight;
    height = 8;
    length = 12;
    width = 10;
    volume = height * length * width;
    weight = (volume + 165) / 166;
    printf("Dimensions: %dx%dx%d\n", length, width, height);
```

```

printf("Volume (cubic inches): %d\n", volume);
printf("Dimensional weight (pounds): %d\n", weight);
return 0;
}

```

L'output del programma è:

```

Dimensions: 12x10x8
Volume (cubic inches): 960
Dimensional weight (pounds): 6

```

Inizializzazione

Alcune delle variabili vengono automaticamente impostate a zero quando un programma inizia l'esecuzione, anche se per la maggior parte non è così [inizializzazione delle variabili > 18.5]. Una variabile che non ha un valore di default e alla quale il programma non ha ancora assegnato un valore è detta non inizializzata.



Tentare di accedere a una variabile non inizializzata (per esempio, stampando il suo valore con una printf o utilizzandola in una espressione) può portare a risultati non predibibili come 2568, -30891 o qualche altro numero ugualmente strano. Con alcuni compilatori possono verificarsi anche comportamenti peggiori – come il blocco del programma.

Naturalmente possiamo sempre dare un valore iniziale a una variabile attraverso il suo assegnamento. C'è una via più semplice però: mettere il valore iniziale della variabile nella sua dichiarazione. Per esempio, possiamo dichiarare la variabile height e inizializzarla in un solo passaggio:

```
int height = 8;
```

Nel gergo del C il valore 8 viene detto **inizializzatore**.

All'interno della stessa dichiarazione può essere inizializzato un qualsiasi numero di variabili:

```
int height = 8, length = 12, width = 10;
```

Tenete presente che ogni variabile richiede il suo inizializzatore. Nell'esempio seguente l'inizializzatore 10 è valido solo per la variabile width e non per le variabili height o length (che rimangono non inizializzate):

```
int height, length, width = 10;
```

Stampare espressioni

La printf non si limita a stampare i valori memorizzati all'interno delle variabili, può anche visualizzare il valore di una qualsiasi espressione numerica. Trarre vantaggio di questa proprietà può semplificare un programma e ridurre il numero di variabili. Per esempio, le istruzioni di pagina seguente

```
volume = height * length * width;
printf("%d\n", volume);
```

possono essere sostituite con

```
printf("%d\n", height * length * width);
```

l'abilità della printf di stampare espressioni illustra uno dei principi generali del C: ovunque venga richiesto un valore può essere utilizzata un'espressione che sia dello stesso tipo.

2.5 Leggere l'input

Considerato che il programma dweight.c calcola il peso dimensionale di solo un pacco, non è particolarmente utile. Per migliorare il programma abbiamo bisogno di permettere all'utente di immettere le dimensioni del pacco.

Per ottenere l'input immesso dall'utente utilizzeremo la funzione scanf, la controparte della printf nella libreria del C. La f di scanf, come la f di printf, sta per "formattato": sia scanf che printf richiedono l'uso di una **stringa di formato** per specificare come deve apparire l'input o l'output dei dati. La scanf ha bisogno di sapere che forma prenderanno i dati di input, così come la printf ha bisogno di sapere come stampare i dati nell'output.

Per leggere un valore int useremo la scanf in questo modo:

```
scanf("%d", &i); /* legge un intero e lo memorizza dentro i */
```

La stringa "%d" dice alla scanf di leggere un input che rappresenta un intero mentre i è una variabile int nella quale vogliamo che scanf memorizzi il valore in ingresso. Il simbolo & è difficile da spiegare a questo punto della trattazione [operatore & >11.2]. Per ora vi farò soltanto notare che di solito (ma non sempre) è necessario quando si usa la scanf.

Leggere un valore float richiede una chiamata alla scanf leggermente diversa:

```
scanf("%f", &x); /* legge un valore float e lo memorizza dentro x */
```

l'operatore %f funziona solo con le variabili di tipo float, così mi assicuro che x si una variabile di tipo float. La stringa "%f" dice alla scanf di cercare un valore di input nel formato dei valori float (il numero può contenere la virgola, anche se questo non è strettamente necessario).

PROGRAMMA

Calcolare il peso dimensionale di un pacco (rivisitato)

Ecco la versione migliorata del programma per il peso dimensionale, dove l'utente può immettere le dimensioni del pacco. Notate che ogni chiamata della scanf è immediatamente preceduta da una chiamata della printf. In questo modo l'utente saprà quando e quali dati deve immettere:

```
dweight2.c /* Calcola il peso dimensionale di un pacco dall'input dell'utente */
#include <stdio.h>
```

```

int main(void)
{
    int height, length, width, volume, weight;

    printf("Enter height of box: ");
    scanf("%d", &height);
    printf("Enter length of box: ");
    scanf("%d", &length);
    printf("Enter width of box: ");
    scanf("%d", &width);
    volume = height * length * width;
    weight = (volume + 165) / 166;

    printf("Volume (cubic inches): %d\n", volume);
    printf("Dimensional weight (pounds): %d\n", weight);

    return 0;
}

```

L'output del programma si presenta in questo modo (l'input immesso dall'utente è sottolineato)

```

Enter height of box: 8
Enter length of box: 12
Enter width of box: 10
Volume (cubic inches): 960
Dimensional weight (pounds): 6

```

Un messaggio che chiede all'utente di immettere dell'input (un cosiddetto **prompt**) normalmente non dovrebbe finire con un carattere new-line perché vogliamo che l'utente immetta l'input sulla stessa riga del prompt stesso. Quando l'utente preme il tasto Invio, il cursore si muoverà automaticamente sulla nuova riga – quindi il programma non ha bisogno di stampare un carattere new-line per terminare la riga corrente.

Il programma `dweight2.c` è affetto da un problema: non lavora correttamente se l'utente immette dell'input non numerico. La Sezione 3.2 discuterà di questo problema in maggiore dettaglio.

2.6 Definire nomi e costanti

Quando un programma contiene delle costanti è una buona pratica assegnarvi dei nomi. I programmi `dweight.c` e `dweight2.c` si basano sulla costante 166, il cui significato potrebbe non apparire chiaro a qualcuno che legga il programma in un secondo momento. Utilizzando la funzionalità detta **definizione di una macro** possiamo dare a questa costante un nome:

```
#define INCCHES_PER_POUND 166
```

`#define` è una direttiva del preprocessore, proprio come lo è `#include`, per questo motivo non c'è nessun punto e virgola alla fine della riga.

Quando il programma viene compilato, il preprocessore rimpiazza ogni macro con il valore che rappresenta. Per esempio, l'istruzione

```
weight = (volume + INCHES_PER_POUND - 1) / INCHES_PER_POUND;
```

diventerà

```
weight = (volume + 166 - 1) / 166;
```

che ha lo stesso effetto che avremmo avuto scrivendo direttamente la seconda riga.

Il valore di una macro può anche essere un'espressione:

```
#define RECIPROCAL_OF_PI (1.0f / 3.12159f)
```

Se contiene degli operatori l'espressione dovrebbe essere racchiusa tra parentesi [parentesi nelle macro > 14.3].

Ponete attenzione al fatto che abbiamo usato solo lettere maiuscole nei nomi della macro. Questa è una convenzione che molti programmati C seguono, non una richiesta del linguaggio (i programmati C lo hanno fatto comunque per decenni, non dovreste essere proprio voi i primi a dissociarvi da questa pratica).

PROGRAMMA

Convertire da Fahrenheit a Celsius

Il programma seguente chiede all'utente di inserire una temperatura espressa in gradi Fahrenheit e poi scrive il suo equivalente in gradi Celsius. L'output del programma avrà il seguente aspetto (l'input immesso dall'utente è sottolineato):

Enter Fahrenheit temperature: 212

Celsius equivalent: 100.0

Il programma accetterà temperature non intere. Questo è il motivo per cui la temperatura Celsius viene stampata come 100.0 invece che 100. Per prima cosa diamo un'occhiata all'intero programma, successivamente vedremo come è strutturato.

```
celsius.c /* Converte una temperatura Fahrenheit in Celsius */
#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void)
{
    float fahrenheit, celsius;
    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);
    celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
    printf("Celsius equivalent: %.1f\n", celsius);
    return 0;
}
```

La riga

```
celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
```

converte la temperatura Fahrenheit in Celsius. Dato che FREEZING_PT sta per 32.0f e SCALE_FACTOR sta per (5.0f / 9.0f), il compilatore vede questa linea come se ci fosse scritto

```
celsius = (fahrenheit - 32.0f) * (5.0f / 9.0f);
```

Definire SCALE_FACTOR come (5.0f / 9.0f) invece che (5 / 9) è importante perché il C tronca il risultato della divisione tra due numeri interi. Il valore (5 / 9) equivalebbe a 0, che non è assolutamente quello che vogliamo.

La chiamata alla printf scrive la temperatura Celsius:

```
printf("Celsius equivalent: %.1f\n", celsius);
```

Notate l'utilizzo di %.1f per visualizzare una sola cifra dopo il separatore decimale.

2.7 Identificatori

Quando scriviamo un programma dobbiamo scegliere un nome per le variabili, le funzioni, le macro e le altre entità. Questi nomi vengono chiamati **identificatori**. In C un identificatore può contenere lettere, cifre e *underscore* ma deve iniziare con una lettera o con un underscore (in C99 gli identificatori possono contenere anche gli *universal character names [universal character names > 25.4]*).

Di seguito alcuni esempi di possibili identificatori:

```
times10 get_next_char _done
```

I seguenti identificatori, invece, non sono ammessi:

```
10items get-next-char
```

L'identificatore 10times inizia con una cifra, non con una lettera o un underscore. get-next-char invece contiene il segno meno e non degli underscore.

Il C è **case-sensitive**, distingue tra caratteri maiuscoli e minuscoli all'interno degli identificatori. Per esempio, i seguenti identificatori sono considerati differenti:

```
job joB j0b j0B Job JoB J0b J0B
```

Gli otto identificatori possono essere utilizzati tutti simultaneamente, ognuno per uno scopo completamente diverso (ricordate l'offuscamento del codice!). I programmati-ri più accorti cercano di far apparire diversi gli identificatori a meno che non siano in qualche modo correlati.

Considerato che nel C la differenza tra maiuscole e minuscole è importante, molti programmati-ri cercano di seguire la convenzione di utilizzare solo le lettere minuscole negli identificatori (che non siano delle macro), inserendo degli underscore ove necessario per la leggibilità:

```
symbol_table current_page name_and_address
```

Altri programmatore evitano gli underscore e utilizzano una lettera maiuscola per iniziare ogni parola all'interno dell'identificatore:

symbolTable currentPage nameAndAddress

(a volte anche la prima lettera viene posta in maiuscolo). Sebbene il primo stile sia comune nel C tradizionale, il secondo sta diventando più popolare grazie alla larga diffusione dell'uso di Java e del C# (e meno diffusamente nel C++). Esistono anche altre convenzioni altrettanto ragionevoli, in ogni caso la cosa importante è che vi assicurate di utilizzare sempre la stessa combinazione di maiuscole e minuscole quando vi riferite allo stesso identificatore.



Il C non pone alcun limite sulla lunghezza degli identificatori, quindi non abbiate paura di utilizzare nomi lunghi e descrittivi. Un nome come `current_page` è molto più facile da capire rispetto a un nome come `cp`.

Keyword



Le parole chiave (**keyword**) della Tabella 2.1 hanno un significato speciale per i compilatori C e quindi non possono essere utilizzate come identificatori. Notate che cinque delle keyword sono state aggiunte in C99.

Tabella 2.1 Keyword

auto	enum	restrict [†]	unsigned
break	extern	return	void
case	float	short	volatile
char	for	signed	while
const	goto	sizeof	_Bool [†]
continue	if	static	_Complex [†]
default	inline [†]	struct	_Imaginary [†]
do	int	switch	
double	long	typedef	
else	register	union	

[†]solo C99

A causa del fatto che il C è case-sensitive, le parole chiave devono apparire esattamente come appaiono in Tabella 2.1, ovvero con tutte le lettere minuscole. Anche i nomi delle funzioni della libreria standard (come la `printf`) contengono solo lettere minuscole. Evitate la triste condizione dello sfortunato programmatore che scrive un intero programma in lettere maiuscole solo per scoprire che il compilatore non può riconoscere le keyword e le chiamate alle funzioni di libreria.



Fate attenzione ad altre restrizioni sugli identificatori. Alcuni compilatori trattano certi identificativi come keyword aggiuntive (asm, per esempio). Anche gli identificatori che appartengono alla libreria standard sono vietati allo stesso modo. Utilizzare uno di questi nomi può causare un errore durante la compilazione o le operazioni di linking. Anche gli identificatori che iniziano per underscore sono riservati [restrizioni sugli identificatori > 21.1].

2.8 La stesura di un programma C

Possiamo pensare a un programma C come a una serie di token: ovvero gruppi di caratteri che non possono essere separati tra loro senza cambiarne significato. Gli identificatori e le keyword sono dei token. allo stesso modo lo sono gli operatori come + e -, i segni di interpunkzione come la virgola e il punto e virgola, e le stringhe letterali. Per esempio, l'istruzione

```
printf("Height: %d\n", height);
```

consiste di sette token:

```
printf (      "Height: %d\n" ,      height ) ;  
① ②       ③       ④       ⑤   ⑥   ⑦
```

I token ① e ⑤ sono identificatori, il token ③ è una stringa letterale, mentre i token ②, ④, ⑥ e ⑦ sono segni di interpunkzione.

Nella maggior parte dei casi la quantità di spazio tra i token presenti all'interno dei programmi non è importante. A un estremo i token possono venire ammucchiati senza spazio tra essi, a eccezione dei punti dove questo causerebbe la fusione di due token formandone un terzo. Per esempio, noi potremmo eliminare la maggior parte dello spazio nel programma celsius.c della Sezione 2.6 lasciando solo lo spazio tra i token come int e main e tra float e fahrenheit:

```
/* Converte una temperatura Fahrenheit in Celsius*/  
#include <stdio.h>  
#define FREEZING_PT 32.0f  
#define SCALE_FACTOR (5.0f / 9.0f)  
int main(void){float fahrenheit, celsius;printf("Enter Fahrenheit temperature:  
");scanf("%f", &fahrenheit);celsius=(fahrenheit-FREEZING_PT)*SCALE_FACTOR;  
printf("Celsius equivalent: %.1f\n", celsius);return 0;}
```

In effetti, se la pagina fosse stata più larga, avremmo potuto scrivere l'intera funzione main su una singola riga. Tuttavia mettere l'intero *programma* su una riga non è possibile perché ogni direttiva del preprocessore ne richiede una separata.

Comprimere i programmi in questo modo non è affatto una buona idea. Aggiungere spazi e linee vuote a un programma lo rende più facile da leggere e capire. Fortunatamente il C permette di inserire una quantità qualsiasi di spazio (spazi vuoti, tabulazioni e caratteri new-line) in mezzo ai token. Questa regola ha delle conseguenze sulla stesura di un programma.

- **Le istruzioni possono essere suddivise** su un qualsivoglia numero di righe. La seguente istruzione per esempio è così lunga che sarebbe difficile comprimerla in una singola riga:

```
printf("Dimensional weight (pounds): %d\n",  
      (volume + INCHES_PER_POUND - 1) / INCHES_PER_POUND);
```

- **Lo spazio tra i token rende più facile all'occhio umano la loro separazione.** Per questa ragione di solito metto uno spazio prima e dopo di ogni operatore:

```
volume = height * length * width;
```

Metto inoltre uno spazio dopo ogni virgola. Alcuni programmati si spingono oltre mettendo spazi anche attorno alle parentesi e ad altri segni di interpuzione.

D&R

- **L'indentazione rende più facile l'annidamento.** Per esempio potremmo indentare le dichiarazioni e le istruzioni per rendere chiaro che sono annidate all'interno del `main`.
- **Le righe vuote possono dividere il programma in unità logiche, rendendo più facile al lettore la comprensione della struttura del programma.** Un programma senza righe vuote è difficile da leggere esattamente come lo sarebbe come un libro senza capitoli.

Il programma `celsius.c` della Sezione 2.6 mette in pratica diverse di queste linee guida. Diamo un'occhiata più attenta alla funzione `main` di quel programma:

```
int main(void)
{
    float fahrenheit, celsius;

    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);

    celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;

    printf("Celsius equivalent: %.1f\n", celsius);

    return 0;
}
```

Per prima cosa osservate come lo spazio attorno a `=`, `-` e `*` faccia risaltare questi operatori. Secondo, notate come l'indentazione delle dichiarazioni e delle istruzioni renda ovvia la loro appartenenza al `main`. Osservate infine come le righe vuote dividano il `main` in cinque parti: (1) dichiarazione delle variabili `fahrenheit` e `celsius`, (2) ottenimento della temperatura `Fahrenheit`, (3) calcolo del valore di `celsius`, (4) stampa della temperatura `Celsius` e (5) ritorno del controllo al sistema operativo.

Visto che stiamo trattando l'argomento del layout di un programma, fate attenzione a com'è ho posizionato sotto `main()` il token `{` e a come ho allineato il token `}` corrispondente. Mettere il token `}` su una riga separata ci permette di inserire o cancellare istruzioni alla fine di una funzione. Inoltre allinearla con `{` rende più facile l'individuazione della fine del `main`.

Una nota finale: sebbene spazio extra possa essere aggiunto *in mezzo* ai token, non è possibile aggiungere spazio dentro un token senza cambiare il significato del programma o causare un errore. Scrivere

```
fl oat fahrenheit, celsius;      /*** SBAGLIATO ***/
oppure
fl
oat fahrenheit, celsius; /*** SBAGLIATO ***/
```

produce un errore mentre il programma viene compilato. Mettere uno spazio all'interno di una stringa è permesso, tuttavia cambia il significato della stringa stessa. Non è consentito però inserire un carattere di new-line all'interno di una stringa (in altre parole spezzando la stringa su due righe):

```
printf("To C, or not to C:  
that is the question.\n");      /*** SBAGLIATO ***/
```

Protrarre una stringa sulle righe successive richiede una speciale tecnica che impareremo più avanti nel testo [[continuare una stringa > 13.1](#)].

Domande & Risposte

D: Cosa significa GCC? [p. 13]

R: Originariamente GCC stava per "GNU C Compiler". Adesso è l'abbreviazione per "GNU Compiler Collection" perché la versione corrente di GCC compila programmi scritti in diversi linguaggi, inclusi Ada, C, C++, Fortran, Java e Objective-C.

D: D'accordo, allora cosa significa GNU?

R: GNU sta per "GNU's Not UNIX!" (che per inciso si pronuncia *guh-NEW*). GNU è un progetto della Free Software Foundation, un'organizzazione fondata da Richard M. Stallman come protesta contro le restrizioni dei licenzi sul software UNIX. Secondo quanto dice il suo sito web, la Free Software Foundation crede che gli utenti dovrebbero essere liberi di "eseguire, copiare, distribuire, studiare, cambiare e migliorare" il software. Il progetto GNU ha riscritto da zero larga parte del software tradizionale UNIX e lo ha reso disponibile gratuitamente.

GCC ed altri software GNU sono delle componenti fondamentali per Linux. Linux è solo il "kernel" del sistema operativo (la parte che gestisce la schedulazione dei programmi e i servizi base di I/O), mentre il software GNU è necessario per avere un sistema operativo pienamente funzionale. Per ulteriori informazioni sul progetto GNU visitate il sito www.gnu.org.

D: In ogni caso qual è l'importanza del GCC?

R: GCC è importante per diverse ragioni, senza contare il fatto che è gratuito ed è in grado di compilare un gran numero di linguaggi. Funziona su molti sistemi operativi e genera codice per diverse CPU, incluse tutte quelle maggiormente utilizzate. GCC è il compilatore principale per molti sistemi operativi basati su UNIX, inclusi Linux, BSD e Mac OS X ed è utilizzato estensivamente nello sviluppo di software commerciale. Per maggiori informazioni su GCC visitate www.gcc.gnu.org.

D: Quanto è accurato GCC nel trovare gli errori nei programmi?

R: GCC ha varie opzioni a riga di comando che determinano con quanta accuratezza il compilatore debba controllare i programmi. Quando queste opzioni vengono utilizzate il GCC è piuttosto efficace nel trovare i punti potenzialmente problematici presenti all'interno di un programma. Qui ci sono alcune delle opzioni più popolari:

-Wall	Fa in modo che il compilatore produca messaggi di warning quando rileva possibili errori. (-W può essere seguito dai codici per degli specifici warning, -Wall significa "tutte le opzioni -W"). Dovrebbe essere utilizzata congiuntamente con -O per avere il massimo effetto.
-W	Emette dei messaggi di warning addizionali oltre a quelli prodotti da -Wall.
-pedantic	Emette tutti i warning richiesti dal C standard. Causa il rifiuto di tutti i programmi che utilizzano funzionalità non standard.
-ansi	Disabilita le funzionalità del GCC che non appartengono allo standard C e abilita le poche funzionalità standard che sono normalmente disabilitate.
-std=c89	Specifica quale versione del C deve essere utilizzata dal compilatore per controllare un programma
-std=c99	

Queste opzioni sono spesso utilizzate in combinazione:

```
%gcc -O -Wall -W -pedantic -ansi -std=c99 -o pun pun.c
```

D: Perché il C è così conciso? Un programma potrebbe essere molto più leggibile se il C utilizzasse begin ed end al posto di { e }, integer al posto di int e così via. [p.14]

R: La leggenda vuole che la brevità dei programmi C sia dovuta all'ambiente che esisteva nei Laboratori Bell al tempo in cui il linguaggio fu sviluppato. Il primo compilatore C girava su un DEC PDP-11 (uno dei primi minicomputer), i programmati utilizzavano *teletype* (essenzialmente una telescrivente collegata a un computer) per scrivere i programmi e stampare i listati. Considerato che le telescriventi sono particolarmente lente (possono stampare solo 10 caratteri al secondo), minimizzare il numero di caratteri in un programma era chiaramente vantaggioso.

D: In alcuni volumi su C, la funzione main termina con exit(0) in luogo di return 0. È la stessa cosa? [p.15]

R: Quando sono presenti all'interno del main, queste due istruzioni sono del tutto equivalenti: entrambe terminano il programma e restituiscono il valore 0 al sistema operativo. Quale utilizzare è solo questione di gusti.

D: Cosa succede se un programma raggiunge la fine della funzione main senza eseguire l'istruzione return? [p.15]

R: L'istruzione return non è obbligatoria, anche se mancasse il programma terminebbe comunque. Nel C89 il valore restituito al sistema operativo non è definito. Nel C99 se il main è dichiarato come int (come nei nostri esempi) il programma restituisce uno 0 al sistema operativo, altrimenti viene restituito un valore non specificato.

D: Il compilatore rimuove completamente i commenti oppure li sostituisce con spazi bianchi?

R: Qualche vecchio compilatore C cancella tutti i caratteri di ogni commento rendendo possibile scrivere

```
a/**/b = 0;
```

e il compilatore lo interpreta come

`ab = 0;`

Secondo lo standard C, tuttavia, il compilatore deve rimpiazzare ogni commento con un singolo spazio bianco e quindi questo trucchetto non funziona. Ci ritroveremmo invece con la seguente istruzione (non consentita):

`a b = 0;`

D: Come posso capire se il mio programma ha un commento non terminato correttamente?

R: Se siete fortunati il vostro programma non verrà compilato perché il commento lo ha fatto diventare "illegale". Se invece il programma viene compilato, ci sono diverse tecniche che potete utilizzare. Verificare attentamente il programma con un *debugger* rivelerà se qualche riga è stata omessa. Alcuni IDE visualizzano i commenti con un colore particolare per distinguerli dal codice circostante. Se state utilizzando uno di questi ambienti potete individuare facilmente i commenti non terminati dato che le linee di codice che sono state incluse accidentalmente in un commento si troveranno ad avere un colore diverso. Anche un programma come `lint` può essere di aiuto [[lint > 1.2](#)].

D: È ammesso annidare un commento all'interno di un altro?

R: I commenti nel vecchio stile (`/* ... */`) non possono essere annidati. Per esempio, il seguente codice non è ammesso:

```
/*
 *** WRONG ***
*/
```

Il simbolo `*/` nella seconda riga si accoppia con il simbolo `/*` della prima e quindi il compilatore segnalerà come errore il simbolo `*/` presente nella terza riga.

Il divieto del C verso i commenti annidati a volte può essere un problema. Supponete di aver scritto un lungo programma contenente molti commenti. Per disabilitare una porzione del programma temporaneamente (diciamo durante il *testing*) il nostro primo istinto sarebbe quello di "commentare" le righe interessate con `/*` e `*/`. Sfortunatamente questo metodo non funziona se le righe contengono dei commenti vecchio stile. I commenti C99 (quelli che iniziano con `//`) possono anche essere annidati all'interno di commenti scritti nel vecchio stile – un altro vantaggio dell'utilizzare il nuovo tipo di commenti.

In ogni caso c'è un modo migliore per disabilitare porzioni di un programma e lo vedremo più avanti [[disabilitare codice > 14.4](#)].

D: Da dove prende il nome il tipo float? [p. 19]

R: `float` è l'abbreviazione di *floating point*, una tecnica per memorizzare i numeri dove la virgola decimale è "mobile". Un valore `float` tipicamente viene memorizzato in due parti: la frazione (o mantissa) e l'esponente. Il numero `12.0` può essere memorizzato come `1.5 x 23`, per esempio, dove `1.5` è la mantissa e `3` è l'esponente. Qualche linguaggio di programmazione chiama questo tipo `real` invece che `float`.

D: Perché le costanti a virgola mobile necessitano della lettera f? [p.20]

R: Per la spiegazione completa guardate il Capitolo 7. Ecco la risposta breve: una costante che contiene il punto decimale, ma non termina per f, ha come tipo il double (abbreviazione per “double precision”), i valori double vengono memorizzati con maggiore accuratezza rispetto ai valori float. In più i valori double possono essere più grandi rispetto ai float, che è il motivo per cui abbiamo bisogno di aggiungere la lettera f quando facciamo l’assegnamento a una variabile float. Senza la f si potrebbe generare un warning poiché un numero da memorizzare in una variabile float potrebbe eccedere la capacità di quest’ultima.

D*: È del tutto vero che non c’è limite nella lunghezza di un identificatore? [p.28]

(C99) **R:** Si e no. Lo standard C89 dice che gli identificatori possono essere arbitrariamente lunghi. Tuttavia ai compilatori è richiesto di ricordare solo i primi 31 caratteri (63 nel C99). Quindi, se due nomi iniziano con gli stessi 31 caratteri un compilatore potrebbe non essere in grado di distinguerli tra loro.

(C99) A rendere le cose ancora più complicate ci sono le regole speciali degli identificatori con linking esterno: la maggior parte dei nomi di funzione ricadono in questa categoria [[linking esterno > 18.2](#)]. Dato che questi nomi devono essere resi noti al linker, e siccome qualche vecchio linker può gestire solo nomi brevi, si ha che nel C89 solamente i primi sei caratteri sono significativi. Inoltre, non dovrebbe essere rilevante che le lettere siano maiuscole o minuscole; di conseguenza, ABCDEFG e abcdefg possono essere trattati come lo stesso nome. (In C99 sono significativi i primi 31 caratteri e la differenza tra maiuscole e minuscole viene presa in considerazione).

La maggior parte dei compilatori e dei linker sono più generosi rispetto allo standard, così queste regole non sono un problema nella pratica. Non preoccupatevi di fare degli identificatori troppo lunghi – preoccupatevi piuttosto di non farli troppo corti.

D: Quanti spazi devo utilizzare per l’indentazione? [p.30]

R: Questa è una domanda difficile. Lasciate troppo poco spazio e l’occhio avrà problemi nell’individuare l’indentazione. Lasciatene troppo e le righe di codice usciranno dallo schermo (o dalla pagina). Molti programmatore C indentano le istruzioni nidificate con otto spazi (un tab), il che è probabilmente eccessivo. Alcuni studi hanno dimostrato che l’ammontare ottimo per l’indentazione è di tre spazi, ma molti programmatore non si sentono a loro agio con numeri che non sono potenze di due. Sebbene di solito io preferisca indentare con tre o quattro spazi, in questo libro utilizzerò due spazi per fare in modo che i programmi rientrino all’interno dei margini.

Esercizi.

Esercizio 2.1

1. Create ed eseguite il famoso programma di Kernighan e Ritchie “hello, world”:

```
#include <stdio.h>
int main(void) {
    printf("hello, world\n");
}
```

Ottenete un messaggio di warning dal compilatore? Se è così, di cosa c’è bisogno per farlo scomparire?

Sezione 2.2 2. *Considerate il seguente programma:

(W)

```
#include <stdio.h>
int main(void) {
    printf("Parkinsons Law:\nWork expands so as to "); printf("fill the time\n");
    printf("available for its completion.\n"); return 0;
}
```

- Identificate le direttive e le istruzioni del programma.
- Che output viene prodotto dal programma?

Sezione 2.4 3. Condensate il programma dweight.c (1) rimpiazzate gli assegnamenti a height, length e width con delle inizializzazioni, (2) rimuovete la variabile weight e al suo posto calcolate $(volume + 165)/166$ all'interno dell'ultima printf.

(W)

- Scrivete un programma che dichiari diverse variabili int e float – senza inizializzarle – e poi stampate i loro valori. C'è qualche schema nei loro valori? (Tipicamente non ce n'è).

Sezione 2.7 5. Quali dei seguenti identificatori non sono ammessi nel C?

(W)

- 100_bottles
 - _100_bottles
 - one_hundred_bottles
 - bottles_by_the_hundred
6. Perché scrivere più caratteri di underscore (come in current__balance, per esempio) adiacenti non è una buona idea?
7. Quali tra le seguenti sono delle parole chiave del C?

- for
- If
- main
- printf
- while

Sezione 2.8 8. Quanti token sono presenti nella seguente istruzione?

(W)

```
answer=(3*q-p*p)/3;
```

- Inserite degli spazi tra i token dell'Esercizio 8 per rendere l'istruzione più facile da leggere.
- Nel programma dweight.c (Sezione 2.4) quali spazi sono essenziali?

Progetti di programmazione

- Scrivete un programma che utilizzi la printf per stampare la seguente immagine sullo schermo:

```
*
*
*
* * * *
```

2. Scrivete un programma che calcoli il volume di una sfera con un raggio di 10 metri utilizzando la formula $V=4/3\pi r^3$. Scrivete la frazione $4/3$ come $4.0f/3.0f$ (provate a scriverlo come $4/3$, cosa succede?) Suggerimento: il C non possiede un operatore esponenziale, quindi per calcolare r^3 avrete la necessità di moltiplicare r più volte per se stesso.
3. Modificate il programma del Progetto di programmazione 2 in modo che chieda all'utente di inserire il raggio della sfera.
4. Scrivete un programma che chieda all'utente di inserire un importo in dollari e centesimi e successivamente lo stampi con un addizionale del 5% di tasse:

Enter an amount: 100.00

With tax added: \$105.00

5. Scrivete un programma che chieda all'utente di inserire un valore per x e poi visualizzi il valore del seguente polinomio:

$$3x^5 + 2x^4 - 5x^3 - x^2 + 7x - 6$$

Suggerimento: Il C non ha l'operatore esponenziale, per questo avrete bisogno di moltiplicare x per se stesso ripetutamente per poter calcolare le potenze di x . (Per esempio $x * x * x$ è x elevato al cubo.)

6. Modificate il programma del Progetto di programmazione 5 in modo che il polinomio venga calcolato utilizzando la seguente formula:

$$(((3x + 2)x - 5)x - 1)x + 7)x - 6$$

Notate che il programma modificato esegue meno moltiplicazioni. Questa tecnica per calcolare i polinomi è conosciuta come la regola di Horner.

7. Scrivete un programma che chieda all'utente di inserire un importo in dollari e poi mostri come pagarlo utilizzando il minor numero di biglietti da 20\$, 10\$, 5\$ e 1\$:

Enter a dollar amount: 93

\$20 bills: 4

\$10 bills: 1

\$5 bills: 0

\$1 bills: 3

Consiglio: Dividete la somma per 20 per determinare il numero di biglietti da \$20 dollari necessari e dopo riducete l'ammontare del valore totale dei biglietti da 20\$. Ripetete lo stesso procedimento per i biglietti delle altre taglie. Assicuratevi di usare valori interi e non a virgola mobile.

8. Scrivete un programma che calcoli il saldo rimanente di un prestito dopo il primo, il secondo e il terzo pagamento mensile.

Enter amount of loan: 20000.00

Enter interest rate: 6.0

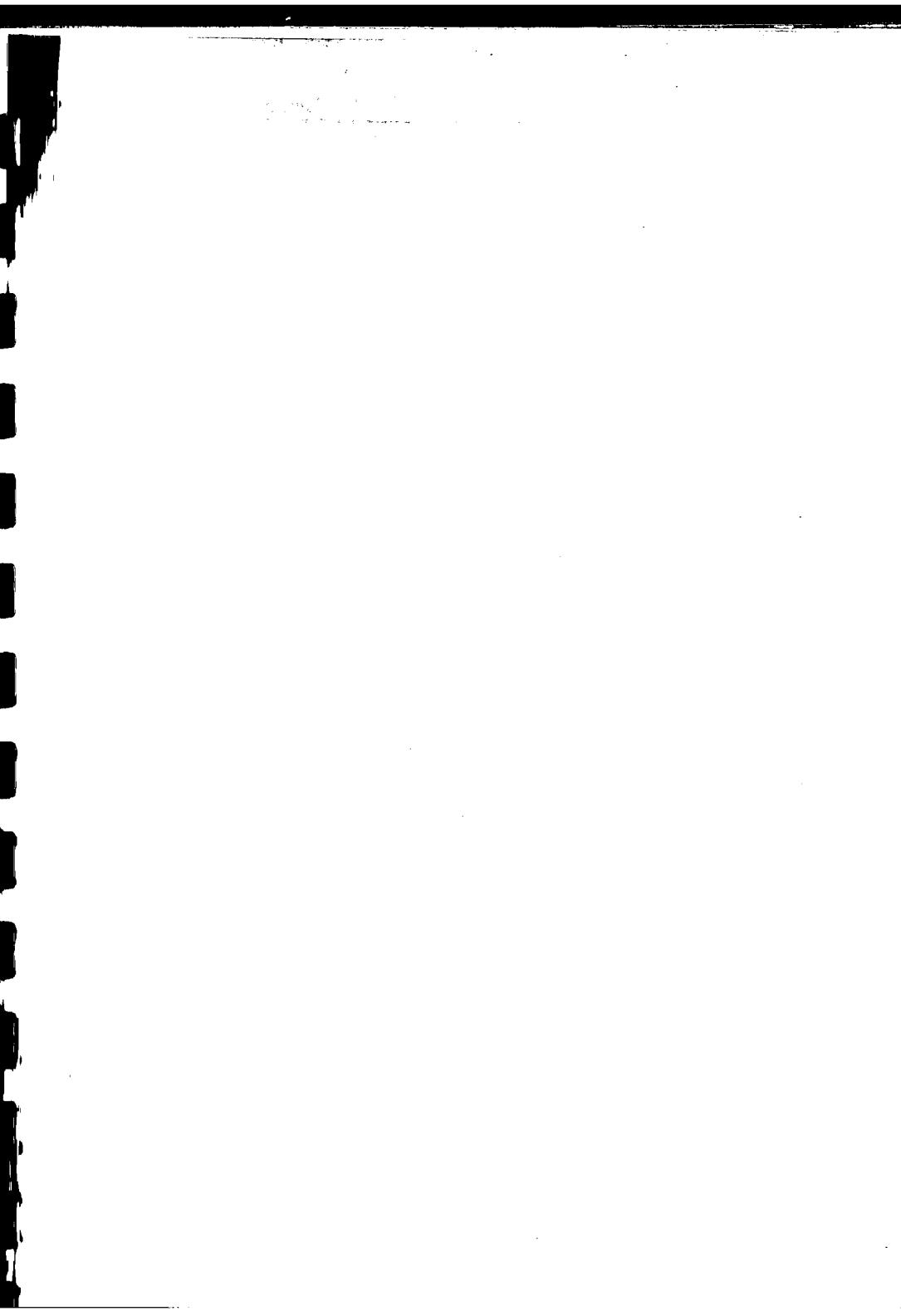
Enter monthly payment: 386.66

Balance remaining after first payment: \$19713.34

Balance remaining after second payment: \$19425.25

Balance remaining after third payment: \$19135.71

Visualizzate ogni saldo con due cifre decimali. Suggerimento: ogni mese il saldo viene decrementato dell'ammontare del pagamento, ma viene incrementato del valore del saldo moltiplicato per la rata mensile di interesse. Per trovare la rata mensile di interesse convertite il tasso d'interesse immesso dall'utente in un numero percentuale e dividetelo per 12.



3 Input/Output formattato

`scanf` e `printf` consentono la lettura e la scrittura formattata e sono due delle funzioni utilizzate più di frequente in C. Questo capitolo illustra come entrambe siano potenti ma al contempo difficili da utilizzare in modo appropriato. La Sezione 3.1 descrive la funzione `printf`, la Sezione 3.2 invece tratta la funzione `scanf`. Per una trattazione più completa si veda il Capitolo 22.

3.1 La funzione `printf`

La funzione `printf` è progettata per visualizzare il contenuto di una stringa, conosciuta come **stringa di formato**, assieme a valori inseriti in specifici punti della stringa stessa. Quando viene invocata, alla `printf` deve essere fornita la stringa di formato seguita dai valori che verranno inseriti durante la stampa:

```
printf(string, espr1, espr2, ...);
```

I valori visualizzati possono essere costanti, variabili oppure espressioni più complicate. Non c'è limite al numero di valori che possono essere stampati con una singola chiamata alla `printf`.

La stringa di formato può contenere sia caratteri ordinari che **specifiche di conversione** che iniziano con il carattere %. Una specifica di conversione è un segnaposto rappresentante un valore che deve essere inserito durante la stampa. L'informazione che segue il carattere % specifica come il valore debba essere convertito dalla sua forma interna (binaria) alla forma da stampare (caratteri) (da questo deriva il termine "specifica di conversione"). Per esempio, la specifica di conversione %d indica alla `printf` che deve convertire un valore int dalla rappresentazione binaria a una stringa di cifre decimali, mentre %f fa lo stesso per i valori float.

Nelle stringhe di formato, i caratteri ordinari vengono stampati esattamente come appaiono, mentre le specifiche di conversione vengono rimpiazzate dal valore che deve essere stampato. Considerate l'esempio di pagina seguente:

```

int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892f;
y = 5527.0f;

printf("i = %d, j = %d, x = %f, y = %f\n", i, j, x, y);

```

Questa chiamata alla printf produce il seguente output:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

I caratteri ordinari nella stringa di formato vengono semplicemente copiati nella riga di output. Le quattro specifiche di conversione vengono sostituite dai valori delle variabili i, j, x e y.



Ai compilatori C non viene richiesto di controllare se il numero di specifiche di conversione presenti in una stringa di formato corrisponda al numero di oggetti di output. La seguente chiamata alla printf ha un numero di specifiche maggiore di quello dei valori da stampare:

```
printf("%d %d\n", i);      /*** ERRATO ***/
```

la printf stamperà il valore di i correttamente dopodiché visualizzerà un secondo numero intero questa volta privo di significato. Una chiamata con un numero di specifiche insufficiente presenta un problema analogo:

```
printf("%d\n", i, j);      /*** ERRATO ***/
```

In questo caso la printf stampa il valore di i ma non quello di j.

Inoltre ai compilatori non viene richiesto di controllare che la specifica di conversione sia appropriata all'oggetto che deve essere stampato. Se il programmatore usa una specifica errata il programma produrrà dell'output privo di significato. Considerate la seguente chiamata alla printf dove la variabile int i e la variabile float x sono state scritte nell'ordine sbagliato:

```
printf("%f %d\n", i, x);      /*** ERRATO ***/
```

visto che la printf deve obbedire alla stringa di formato, visualizzerà obbedientemente un valore float seguito da un valore int. Purtroppo risulteranno entrambi senza significato.

Specifiche di conversione

Le specifiche di conversione forniscono al programmatore grandi potenzialità di controllo sull'aspetto dell'output, ma possono rivelarsi complicate e difficili da leggere. Infatti, una descrizione dettagliata delle specifiche di conversione è un compito troppo arduo per essere affrontato a questo punto del libro. Per questo vedremo in sintesi le caratteristiche più importanti. Nel Capitolo 2 abbiamo visto che le specifiche di conversione possono includere informazioni sulla formattazione e, in particolare, abbiamo utilizzato %.1f per stampare un valore float con una sola cifra dopo il separato-

re decimale. Più in generale una specifica di conversione può avere la forma `%om.pX o %m.pX` dove `m` e `p` sono delle costanti intere e `X` una lettera. Sia `m` che `p` sono opzionali. Se `p` viene omessa il punto che separa `m` e `p` viene omesso a sua volta. Nella specifica di conversione `%10.2f`, `m` è 10, `p` è 2 e `X` è `f`. Nella specifica `%10f`, `m` è 10 e `p` (assieme al punto) è mancante, mentre nella specifica `.2f`, `p` è 2 ed `m` non è presente.

Il campo di minimo, `m`, specifica il numero minimo di caratteri che deve essere stampato. Se il valore da stampare richiede meno di `m` caratteri, il valore verrà allineato a destra (in altre parole, dello spazio extra precederà il valore). Per esempio, la specifica `%4d` stamperebbe il numero 123 come `*123`. (In questo capitolo utilizzerò il carattere `*` per rappresentare il carattere spazio). Se il valore che deve essere stampato richiede più di `m` caratteri il campo si espanderà automaticamente fino a raggiungere la grandezza necessaria. Quindi la specifica `%4d` stamperebbe il numero 12345 come `12345` (non viene persa nessuna cifra). Mettere un segno meno davanti a `m` impone l'allineamento a sinistra, la specifica `-%4d` stamperebbe 123 come `123*`.

Il significato della precisione, `p`, non è facilmente descrivibile in quanto dipende dalla scelta di `X`, lo **specificatore di conversione**. `X` indica quale conversione deve essere applicata al valore prima di stamparlo. Le conversioni più comuni per i numeri sono:

- `d` – stampa gli interi nella forma decimale (base 10). Il valore di `p` indica il numero minimo di cifre da stampare (se necessario vengono posti degli zero aggiuntivi all'inizio del numero); se `p` viene omesso si assume che abbia il valore 1 (in altre parole `%d` è lo stesso `.1d`).
- `e` – stampa un numero a virgola mobile nel formato esponenziale (notazione scientifica). Il valore di `p` indica quante cifre devono apparire dopo il separatore decimale (per default sono 6). Se `p` è 0, il punto decimale non viene stampato.
- `f` – stampa un valore a virgola mobile nel formato a "virgola fissa" senza esponente. Il valore di `p` ha lo stesso significato che per lo specificatore `e`.
- `g` – stampa un valore a virgola mobile sia nel formato esponenziale che in quello decimale a seconda della dimensione del numero. Il valore di `p` specifica il numero di cifre significative (non le cifre dopo il separatore decimale) che devono essere visualizzate. A differenza della conversione `f` la conversione `g` non visualizzerà zeri aggiuntivi. Inoltre, se il valore che deve essere stampato non ha cifre dopo la virgola, `g` non stampa il separatore decimale.

Lo specificatore `g` è utile per visualizzare numeri la cui dimensione non può essere predetta durante la scrittura del programma oppure tende a variare considerevolmente per dimensione. Quando viene utilizzato per stampare numeri non troppo grandi e non troppo piccoli, lo specificatore `g` utilizza il formato a virgola fissa; se, al contrario, viene utilizzato con numeri molto grandi o molto piccoli, lo specificatore `g` passa al formato esponenziale in modo che siano necessari meno caratteri. Ci sono molte altre specifiche oltre a `%d`, `%e`, `%f` e `%g`, ne introdurremo alcune nei capitoli a seguire [**specificatori per gli interi > 7.1; specificatori per i float > 7.2; specificatori per i caratteri > 7.3; specificatori per le stringhe > 13.3**]. Per un elenco completo delle specifiche e delle loro potenzialità consultate la Sezione 22.3.

PRIMA MAMMA

Utilizzare la printf per formattare i numeri

Il programma seguente illustra l'uso della printf per stampare i numeri interi e i numeri a virgola mobile in vari formati.

```
1printf /* Stampa valori int e float in vari formati */
2#include <stdio.h>
3
4int main(void)
5{
6    int i;
7    float x;
8
9    i = 40;
10   x = 839.21f;
11
12   printf("|%d|%5d|%-5d|%.3d|\n", i, i, i, i);
13   printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);
14
15   return 0;
16 }
```

I caratteri | nella stringa di formato della printf servono solamente per aiutare a visualizzare quanto spazio occupa ogni numero quando viene stampato. A differenza di % o \ il carattere | non ha alcun significato particolare per la printf. L'output del programma è:

40	40 40	040	
839.210	8.392e+02	839.21	

Guardiamo più da vicino le specifiche di conversione utilizzate in questo programma:

- %d – Stampa i nella forma decimale utilizzando il minimo spazio necessario.
- %5d – Stampa i nella forma decimale utilizzando cinque caratteri. Dato che i richiede solo due caratteri vengono aggiunti tre spazi.
- %-5d – Stampa i nella forma decimale utilizzando un minimo di cinque caratteri. Dato che i non ne richiede cinque, vengono aggiunti degli spazi successivamente al numero (ovvero i viene allineato a sinistra in un campo lungo cinque caratteri).
- %.3d – Stampa i nella forma decimale utilizzando un minimo di cinque caratteri complessivi e un minimo di tre cifre. Dato che i è lungo solo due cifre, uno zero extra viene aggiunto per garantire la presenza di tre cifre. Il numero risultante è lungo solamente tre cifre così vengono aggiunti solo due spazi per un totale di cinque caratteri (i viene allineato a destra).
- %10.3f – Stampa x nel formato a virgola fissa utilizzando complessivamente 10 caratteri con tre cifre decimali. Dato che x richiede solamente sette caratteri (tre prima del separatore decimale, tre dopo il separatore e uno per il separatore decimale stesso) prima di x vengono messi tre spazi.

- `%10.3e` — Stampa `x` nel formato esponenziale utilizzando complessivamente 10 caratteri con tre cifre dopo il separatore decimale. Tuttavia `x` richiede solo nove cifre (incluso l'esponente), così uno spazio precederà `x`.
- `%-10g` — Stampa `x` o nella forma a virgola fissa o nella forma esponenziale utilizzando 10 caratteri complessivi. In questo caso la `printf` sceglie di stampare `x` nel formato a virgola fissa. La presenza del segno meno forza l'allineamento a sinistra, così `x` viene fatto seguire da quattro spazi.

Sequenze di escape

Il codice `\n` che utilizziamo spesso nelle stringhe di formato è chiamato **sequenza di escape**. Le sequenze di escape permettono alle stringhe di contenere dei caratteri che altrimenti causerebbero dei problemi al compilatore, inclusi i caratteri non stampabili (di controllo) e i caratteri che hanno un significato speciale per il compilatore (come `"`). Daremo più avanti un elenco completo delle sequenze di escape [**sequenze di escape > 7.3**], per ora eccone alcuni esempi:

Alert(bell)	<code>\a</code>
Backspace	<code>\b</code>
New line	<code>\n</code>
Tab	<code>\t</code>

Quando queste sequenze di escape appaiono nelle stringhe di formato della `printf`, rappresentano un'azione che deve essere eseguita durante la stampa. Su molti computer stampare `\a` provoca un beep udibile. Stampare `\b` fa sì che il cursore si muova indietro di una posizione. Stampare `\n` fa avanzare il cursore all'inizio della riga successiva. Infine stampare `\t` sposta il cursore al punto di tabulazione successivo.

Una stringa potrebbe contenere un numero qualsiasi di sequenze di escape. Prendete in considerazione il seguente esempio di `printf` nel quale la stringa di formato contiene sei sequenze di escape:

```
printf("Item\tUnit\tPurchase\n\tPrice\tDate\n");
```

Eseguendo questa istruzione verrà stampato un messaggio su due righe:

Item	Unit	Purchase
Price	Date	

Un'altra sequenza di escape molto comune è `\"` che rappresenta il carattere `"`. Il carattere `"` segna l'inizio e la fine di una stringa e quindi non potrebbe apparire al suo interno senza l'utilizzo di questa sequenza di escape. Ecco un esempio:

```
printf("\\"Hello!\\\"");
```

L'istruzione produce il seguente output:

`"Hello!"`

Per inciso non è possibile mettere un singolo carattere \ in una stringa. In tal caso il compilatore lo interpreterebbe automaticamente come l'inizio di una sequenza di escape. Per stampare un singolo carattere \ si devono inserire nella stringa due caratteri \:

```
printf("\\"); /* stampa un carattere \ */
```

3.2 La funzione scanf

Così come la funzione printf stampa l'output secondo uno specifico formato, la scanf legge l'input secondo un particolare formato. Una stringa di formato della scanf, così come una stringa di formato di una printf, può contenere sia caratteri ordinari che specifiche di conversione. Le conversioni ammesse per la scanf sono essenzialmente le stesse che vengono utilizzate dalla printf.

In molti casi una stringa di formato per la scanf conterrà solo specifiche di conversione, così come accade nell'esempio seguente:

```
int i, j;
float x, y;

scanf("%d%d%f%f", &i, &j, &x, &y);
```

Supponete che l'utente immetta il seguente input:

```
1 -20 .3 -4.0e3
```

La scanf leggerà la riga convertendo i suoi caratteri nei numeri che rappresentano e quindi assegnerà i valori 1, -20, 0.3 e -4000.0 rispettivamente a i, j, x e y. Stringhe di formato "completamente compatte" come "%d%d%f%f" sono comuni nelle chiamate alla scanf. Invece accade molto più raramente che stringhe di formato della printf abbiano delle specifiche di conversione adiacenti. La scanf, come la printf, presenta diverse trappole a chi non vi presta attenzione. Quando viene utilizzata la scanf, il programmatore deve controllare che il numero di conversioni di formato combaci esattamente con il numero di variabili in ingresso e che la conversione sia appropriata per la variabile corrispondente (come con la printf, al compilatore non è richiesto di controllare eventuali discrepanze). Un'altra trappola coinvolge il simbolo & che normalmente precede le variabili nella scanf. Non sempre, ma di solito il carattere & è necessario, sarà quindi responsabilità del programmatore ricordarsi di utilizzarlo.



Dimenticarsi di mettere il simbolo & davanti a una variabile in una chiamata alla scanf avrà dei risultati imprevedibili e a volte disastrosi. Un crash del programma è un esito comune. Come minimo il valore letto dall'input non viene memorizzato nella variabile, anzi, la variabile manterrà il suo valore precedente (che potrebbe essere senza significato se alla variabile non è stato dato un valore iniziale). Omettere il simbolo & è un errore estremamente comune, quindi fate attenzione! Qualche compilatore è in grado di individuare tale errore e può generare dei messaggi di warning come "format argument is not a pointer." (Il termine *pointer* viene descritto nel Capitolo 11. Il simbolo & viene utilizzato per creare un puntatore a una variabile.) Se ottenete un messaggio di errore come questo controllate la possibile mancanza di un &.

Chiamare la `scanf` è un modo efficace per leggere dati, ma non ammette errori. Molti programmati professionisti evitano la `scanf` e leggono tutti i dati sotto forma di caratteri e poi li convertono successivamente in forma numerica. Noi utilizzeremo abbastanza spesso la funzione `scanf`, specialmente nei primi capitoli, perché fornisce un modo semplice per leggere i numeri. Siate consapevoli, tuttavia, che molti dei vostri programmi non si comporteranno a dovere nel caso in cui l'utente immetta dei dati non attesi. Come vedremo più avanti [rilevare gli errori nella `scanf` > 22.3] è possibile controllare all'interno del programma se la `scanf` abbia letto con successo i dati richiesti (e se abbia cercato di riprendersi nel caso non vi fosse riuscita). Questi test non sono praticabili nei programmi di esempio di questo libro: aggiungerebbero troppe istruzioni e oscurerebbero i punti chiave degli esempi stessi.

Come funziona la `scanf`

La funzione `scanf` è in grado di fare molto più di quello che abbiamo visto finora. È essenzialmente un funzione di *pattern matching* che cerca di combinare gruppi di caratteri di input con le specifiche di conversione.

Come la funzione `printf`, anche la `scanf` è controllata da una stringa di formato. Quando viene chiamata, la `scanf` inizia a elaborare le informazioni presenti nella stringa partendo dalla sinistra. Per ogni specifica di conversione della stringa di formato, la `scanf` cerca di localizzare nei dati di input un oggetto del tipo appropriato, saltando degli spazi vuoti, se necessario. La `scanf`, quindi, legge l'oggetto fermandosi non appena incontra un carattere che non può appartenere all'oggetto stesso. Se l'oggetto è stato letto con successo la `scanf` prosegue elaborando il resto della stringa di formato. Se un qualsiasi oggetto non viene letto con successo, la `scanf` termina immediatamente senza esaminare la parte rimanente della stringa di formato (o i rimanenti dati di input).

Quando la `scanf` cerca l'inizio di un numero, ignora i caratteri che rappresentano degli spazi vuoti (i caratteri di spazio, le tabulazioni orizzontali e verticali, e i caratteri new-line). Di conseguenza i numeri possono essere messi sia su una singola riga che sparsi su righe diverse. Considerate la seguente chiamata alla `scanf`:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

Supponete che l'utente immetta tre linee di input:

```
1  
-20 .3  
-4.0e3
```

La `scanf` vede una sequenza continua di caratteri:

```
••1•-20••.3•••-4.0e3•
```

(Stiamo utilizzando il simbolo • per rappresentare gli spazi e il simbolo □ per rappresentare il carattere new-line). Dato che quando cerca l'inizio di un numero salta i caratteri di spazio bianco, la `scanf` sarà in grado di leggere correttamente i valori. Nello schema seguente una s sotto un carattere indica che il carattere è stato saltato mentre una r indica che il carattere è stato letto come parte di un oggetto di input:

```
••1•-20•••.3•••-4.0e3•
ssrsrrssssrrssssrrrrr
```

la `scanf` "guarda" al carattere finale new-line senza leggerlo veramente. Questo new-line sarà il primo carattere letto dalla prossima chiamata alla `scanf`.

Che regole segue la `scanf` per riconoscere un intero o un numero a virgola mobile? Quando le viene chiesto di leggere un intero, la `scanf` per prima cosa va alla ricerca di una cifra, di un segno più o di un segno meno. Successivamente legge le cifre fino a quando non incontra un carattere che non corrisponde a una cifra. Quando le viene chiesto di leggere un numero a virgola mobile, la `scanf` va alla ricerca di un segno più o un segno meno (opzionale), seguito da una serie di cifre (possibilmente contenenti il punto decimale), seguita da un esponente (opzionale). Un esponente consiste di una lettera e (o E), di un segno opzionale, e di una o più cifre.

Le conversioni `%e`, `%f` e `%g` sono intercambiabili nell'utilizzo con la `scanf`. Tutti e tre seguono le stesse regole per riconoscere un numero a virgola mobile.

D&R

Quando la `scanf` incontra un carattere che non può essere parte dell'oggetto corrente, allora questo carattere "viene rimesso a posto" per essere letto nuovamente durante la scansione del prossimo oggetto di input o durante la successiva chiamata alla `scanf`. Considerate la seguente impostazione (inevitabilmente patologica) dei nostri quattro numeri:

```
1-20.3-4.0e3•
```

Utilizziamo la stessa chiamata della `scanf`

```
scanf( "%d%d%f%f", &i, &j, &x, &y);
```

e di seguito vediamo come verrebbe elaborato il nuovo input.

- Specifica di conversione: `%d`. Il primo carattere non vuoto è 1, visto che gli interi possono iniziare con un 1, la `scanf` leggerà il prossimo carattere: -. Riconosciuto che - non può apparire all'interno di un intero, la `scanf` memorizza l'1 in i e rimette a posto il carattere -.
- Specifica di conversione: `%d`. La `scanf` legge i caratteri -, 2, 0 e . (punto). Dato che un intero non può contenere il punto decimale, la `scanf` memorizza -20 in j, mentre il carattere . viene rimesso a posto.
- Specifica di conversione: `%f`. La `scanf` legge i caratteri ., 3 e -. Dato che un numero floating-point non può contenere un segno meno dopo una cifra, la `scanf` memorizza 0.3 dentro x mentre il carattere - viene rimesso a posto.
- Specifica di conversione: `%f`. Alla fine la `scanf` legge i caratteri -, 4, ., 0, e, 3 e • (new-line). Dato che un numero floating-point non può contenere un carattere new-line la `scanf` memorizza -4.0×10^3 dentro y e rimette a posto il carattere new-line.

In questo esempio la `scanf` è in grado di combinare ogni specifica presente nella stringa di formato con un oggetto di input. Dato che il carattere new-line non è stato letto, viene lasciato alla prossima chiamata della `scanf`.

Caratteri ordinari nelle stringhe di formato

Il concetto di *pattern-matching* può essere esteso ulteriormente scrivendo stringhe di formato contenenti caratteri ordinari oltre alle specifiche di conversione. L'azione che la scanf esegue quando elabora un carattere ordinario presente in una stringa di formato, dipende dal fatto che questo sia o meno un carattere di spaziatura.

- **Caratteri di spazio bianco.** Quando in una stringa di formato incontra uno o più caratteri di spaziatura consecutivi, la scanf legge ripetutamente tali caratteri dall'input fino a quando non raggiunge un carattere non appartenente alla spaziatura (il quale viene rimesso a posto). Il numero di caratteri di spaziatura nella stringa di formato è irrilevante. Un carattere di spaziatura nella stringa di formato si adatterà a un qualsiasi numero di caratteri di spaziatura dell'input. (Mettere un carattere di spaziatura in una stringa di formato non forza l'input a contenere dei caratteri di spaziatura. Infatti un carattere di spaziatura in una stringa di formato si combina con un numero qualsiasi di caratteri di spaziatura presenti nell'input e questo comprende il caso in cui non ne sono presenti).
- **Altri caratteri.** Quando in una stringa di formato la scanf incontra un carattere non corrispondente a spaziatura, lo confronta con il successivo carattere di input. Se i due combaciano, la scanf scarta il carattere di input e continua l'elaborazione della stringa. Se invece i due caratteri non combaciano, la scanf rimette il carattere diverso nell'input e poi si interrompe senza elaborare ulteriormente la stringa di formato o leggere altri caratteri di input.

Per esempio, supponete che la stringa di formato sia "%d/%d". Se l'input è

•5/*96

la scanf salta il primo carattere di spazio mentre va alla ricerca di un intero. Successivamente fa combaciare il %d con 5, fa combaciare il / con /, salta lo spazio mentre ricerca un ulteriore intero e fa combaciare il %d con 96. Se invece l'input è

•5*/96

La scanf salta uno spazio, associa il %d a 5, poi cerca di combinare il / della stringa di formato con lo spazio presente nell'input. I due non combaciano e quindi la scanf rimette a posto lo spazio. I caratteri •/*96 rimangono nell'input per essere letti dalla prossima chiamata alla scanf. Per ammettere spazi dopo il primo numero dovremmo utilizzare la stringa di formato "%d /%d".

Confondere printf con scanf

Sebbene le chiamate alla scanf e alla printf possano apparire simili, ci sono delle differenze significative tra le due funzioni. Ignorare queste differenze può essere rischioso per la "salute" del vostro programma.

Uno degli errori più comuni è quello di mettere il simbolo & davanti alle variabili in una chiamata printf:

```
printf ("%d %d\n", &i, &j);      /*** ERRATO ***/
```

Fortunatamente questo errore è facilmente identificabile: al posto di i e j, la printf stamperà una coppia di strani numeri.

Dato che normalmente la scanf salta i caratteri di spaziatura quando va alla ricerca dei dati, spesso non c'è la necessità per una stringa di formato di includere altri caratteri oltre alle specifiche di conversione. Assumere erroneamente che la stringa di formato della scanf debba rispecchiare la stringa di formato della printf (un altro errore comune) può essere causa di comportamenti imprevisti. Guardiamo cosa succede quando viene eseguita la seguente chiamata alla scanf:

```
scanf("%d, %d", &i, &j);
```

La scanf per prima cosa cercherà nell'input un intero, il quale verrà memorizzato nella variabile i. La scanf poi cercherà di combinare la virgola con il successivo carattere di input. Se il successivo carattere di input è uno spazio, non una virgola, la scanf terminerà senza leggere il valore di j.



Sebbene le stringhe di formato della printf finiscano spesso con un \n, mettere un carattere new-line alla fine della stringa di formato di una scanf non è una buona idea. Per la scanf un carattere new-line nella stringa di formato è equivalente a uno spazio. Entrambi fanno avanzare la scanf al successivo carattere non corrispondente alla spaziatura. Per esempio, con la stringa di formato "%d\n", la scanf salterebbe i caratteri di spaziatura, leggerebbe un intero e successivamente salterebbe al successivo carattere non di spaziatura. Una stringa di formato come questa può causare il blocco di un programma interattivo nell'attesa dell'immissione da parte dell'utente di un carattere non appartenente alla spaziatura.

PROGRAMMA

Sommare frazioni

Per illustrare l'abilità di pattern-matching della scanf consideriamo il problema della lettura di una frazione immessa dall'utente. Per consuetudine le frazioni vengono scritte nella forma *numeratore/denominatore*. Invece di far immettere all'utente il numeratore e il denominatore separatamente, la scanf rende possibile la lettura di un'intera frazione. Il seguente programma, che fa la somma di due frazioni, illustra questa tecnica.

```
addfrac.c /* Sommare due frazioni */
#include <stdio.h>
int main(void)
{
    int num1, denom1, num2, denom2, result_num, result_denom;
    printf("Enter first fraction: ");
    scanf("%d/%d", &num1, &denom1);
    printf("Enter second fraction: ");
    scanf("%d/%d", &num2, &denom2);
    result_num = num1 * denom2 + num2 * denom1;
    result_denom = denom1 * denom2;
    printf("The sum is %d/%d", result_num, result_denom);
```

```
result_denom = denom1 * denom2;
printf("The sum is %d/%d\n", result_num, result_denom);
return 0;
}
```

Una sessione di questo programma potrebbe presentarsi come segue:

```
Enter first fraction: 5/6
Enter second fraction: 3/4
The sum is 38/24
```

Notate che la frazione prodotta non è ridotta ai minimi termini.

Domande e risposte

D*: Abbiamo visto la conversione %i utilizzata per leggere e scrivere interi. Qual è la differenza tra %i e %d? [p. 41]

R: In una stringa di formato per la printf non c'è nessuna differenza tra le due. In una stringa di formato della scanf però la %d può associarsi solo a numeri scritti in forma decimale (base 10), mentre la %i può associarsi con interi espressi in ottale (base 8 [**numeri ottali > 7.1**]), decimale o esadecimale (base 16 [**numeri esadecimale > 7.1**]). Se un numero di input ha uno zero come prefisso (come 056), la %i lo tratta come un numero ottale. Se il numero ha un prefisso come 0x o 0X (come in 0x56), la %i lo tratta come un numero esadecimale. Utilizzare la specifica %i invece che la %d per leggere un numero può avere dei risultati inaspettati nel caso in cui l'utente dovesse accidentalmente mettere uno 0 all'inizio del numero. A causa di questo inconveniente, vi raccomando vivamente di utilizzare la specifica %d.

D: Se la printf tratta il % come l'inizio di una specifica di conversione, come posso stampare il carattere %?

R: Se la printf incontra due caratteri % consecutivi in una stringa di formato, allora stampa un singolo carattere %. Per esempio l'istruzione

```
printf("Net profit: %d%%\n", profit);
```

potrebbe stampare

Net profit: 10%

D: Il codice di escape \t dovrebbe far procedere la printf al prossimo stop della tabulazione. Come faccio a sapere quanto distante è questo punto? [p.43]

R: Non potete saperlo. L'effetto della stampa di un \t non è definito in C. Infatti dipende da quello che fa il vostro sistema operativo quando gli viene chiesto di stampare un carattere di tabulazione. I punti di stop delle tabulazioni sono tipicamente distanziati di 8 caratteri, ma il C non dà garanzie su questo.

D: Cosa fa la scanf se gli viene chiesto di leggere un numero e l'utente immette un input non numerico?

R: Guardiamo al seguente esempio:

```
printf("Enter a number: ");
scanf("%d", &i);
```

Supponete che l'utente immetta un numero valido seguito da dei caratteri non numerici:

Enter a number: 23foo

In questo caso la scanf legge 2 e 3 memorizzando 23 in i. I caratteri rimanenti (foo) vengono lasciati per essere letti dalla prossima chiamata della scanf (o da qualche altra funzione di input). D'altra parte, supponete che l'input sia non valido dall'inizio:

Enter a number: foo

In questo caso il valore di i non è definito e foo viene lasciato alla prossima scanf.

Cosa possiamo fare per questa spiacevole situazione? Più avanti vedremo come fare a controllare se una chiamata alla scanf ha avuto successo [[rilevare gli errori nella scansione](#) > [22.3](#)]. Se la chiamata non ha buon esito, potremmo far terminare il programma cercare di risolvere la situazione, magari scartando l'input inconsistente e chiedendo all'utente di immettere nuovamente i dati (metodi per scartare dell'input non corretto vengono discussi nella sezione D&R alla fine del Capitolo 22).

D: Non capiamo come la scanf possa rimettere i caratteri letti nell'input affinché questi possano essere letti nuovamente. [p. 46]

Agli effetti pratici i programmi non leggono l'input dell'utente così come quest'ultimo viene digitato. L'input, al contrario, viene memorizzato in un *buffer* nascosto al quale la funzione scanf ha accesso. Per la scanf è semplice rimettere i caratteri nel buffer per renderli disponibili alle letture successive. Il Capitolo 22 discute in maggiore dettaglio del *buffering* dell'input.

D: Cosa fa la scanf se l'utente immette segni di interruzione (delle virgole, per esempio) tra i numeri?

R: Diamo un'occhiata a questo semplice esempio: supponete di dover leggere una coppia di interi utilizzando la scanf:

```
printf("Enter two numbers: ");
scanf("%d%d", &i, &j);
```

Se l'utente immette

4,28

la scanf leggerà 4 e lo memorizzerà all'interno di i. Appena cerca l'inizio del secondo numero, la scanf incontra la virgola. Visto che i numeri non possono iniziare con una virgola, la scanf termina immediatamente. La virgola e il secondo numero vengono lasciati per la prossima chiamata alla scanf.

Naturalmente possiamo risolvere facilmente il problema aggiungendo una virgola alla stringa di formato se siamo sicuri che i numeri saranno *sempre* separati da una virgola.

```
printf("Enter two numbers, separated by a comma: ");
scanf("%d,%d", &i, &j);
```

Esercizi

Sezione 3.1

1. Che output producono le seguenti chiamate alla printf?

- (a) `printf("%6d, %4d", 86, 1040);`
- (b) `printf("%12.5e", 30.253);`
- (c) `printf("%.4f", 83.162);`
- (d) `printf("%-6.2g", .0000009979);`

W 2. Scrivete delle chiamate alla printf per visualizzare la variabile float x nei formati seguenti:

- (a) Notazione esponenziale, allineamento a sinistra in un campo di dimensione 8, una cifra dopo il separatore decimale.
- (b) Notazione esponenziale, allineamento a destra in un campo di dimensione 10, sei cifre dopo il separatore decimale.
- (c) Notazione a virgola fissa, allineamento a sinistra in un campo di dimensione 8, tre cifre dopo il separatore decimale.
- (d) Notazione a virgola fissa, allineamento a destra in un campo di dimensione 6, nessuna cifra dopo il separatore decimale.

Sezione 3.2 3. Per ognuna della seguenti coppie di stringhe di formato della scanf indicate se queste sono equivalenti o meno. Se non lo sono mostrate come possono essere distinte.

- (a) "%d" e "%d"
- (b) "%d-%d-%d" e "%d -%d -%d"
- (c) "%f" e "%f"
- (d) "%f,%f" e "%f, %f"

4. *Supponiamo di chiamare la funzione scanf nel modo seguente:

```
scanf("%d%f%d", &i, &x, &j);
```

Se l'utente immette

10.3 5 6

quali saranno i valori di i, x e j dopo la chiamata? (Assumete che i e j siano variabili int e che x sia una variabile float).

W 5. *Supponiamo di chiamare la funzione scanf come segue:

```
scanf("%f%d%f", &x, &i, &y);
```

Se l'utente immette

12.3 45.6 789

quali saranno i valori di x, i e y dopo la chiamata? (Assumete che x e y siano variabili float e che i sia una variabile int).

* Gli esercizi contrassegnati con un asterisco sono difficili – solitamente la risposta corretta non è quella ovvia. Leggete la domanda attentamente prestando attenzione e riguardando la relativa sezione, se necessario!

6. Modificate il programma `addfrac.c` della Sezione 3.2 in modo che all'utente venga permesso di immettere frazioni che contengano degli spazi prima e dopo il carattere /.

Progetti di programmazione

- W 1. Scrivete un programma che accetti la data dall'utente nella forma `mm/dd/yyyy` e poi stampatela nella forma `yyyy-mm-dd`:

Enter a date (mm/dd/yyyy): 2/17/2011
You entered the date 20110217

2. Scrivete un programma che formatti le informazioni inserite dall'utente. Una sessione del programma deve presentarsi in questo modo:

Enter item number: 583

Enter unit price: 13.5
Enter purchase date (mm/dd/yyyy): 10/24/2010

Item	Unit	Purchase
	Price	Date
583	\$13.50	10/24/2010

Il numero indicante l'articolo e la data d'acquisto devono essere allineati a sinistra mentre il prezzo unitario deve essere allineato a destra. Ammettete somme in dollari fino a 9999,99 \$. Suggerimento: utilizzate le tabulazioni per allineare le colonne.

- W 3. I libri sono identificati da un numero chiamato *International Standard Book Number* (ISBN). I numeri ISBN assegnati dopo il primo gennaio 2007 contengono 13 cifre suddivise in 5 gruppi come 978-0-393-97950-3 (i vecchi numeri ISBN utilizzavano 10 cifre). Il primo gruppo di cifre (il prefisso GS1) correntemente è 978 o 979. Il gruppo successivo specifica la lingua o il Paese di origine (per esempio 0 e 1 sono utilizzati nei Paesi anglofoni). Il *publisher code* identifica l'editore (393 è il codice per la casa editrice W.W. Norton). L'*item number* viene assegnato dall'editore per identificare uno specifico libro (97950 è il codice della versione originale di questo libro). Un ISBN finisce con una cifra di controllo che viene utilizzata per verificare la correttezza delle cifre precedenti. Scrivete un programma che suddivida in gruppi il codice ISBN immesso dall'utente:

Enter ISBN: 978-0-393-97950-3
GS1 prefix: 978
Group identifier: 0
Publisher code: 393
Item number: 97950
Check digit: 3

Nota: il numero di cifre in ogni gruppo può variare. Non potete assumere che i gruppi abbiano sempre la lunghezza presentata in questo esempio. Testate il vo-

stro programma con dei codici ISBN reali (solitamente si trovano nel retro della copertina dei libri e nelle pagine relative ai diritti d'autore).

4. Scrivete un programma che chieda all'utente di inserire un numero telefonico nella forma (xxx) xxx-xxxx e successivamente stampi il numero nella forma xxx.xxxx:

Enter phone number [(xxx) xxx-xxxx]: (404) 817-6900
You entered 404.817.6900

5. Scrivete un programma che chieda all'utente di inserire i numeri da 1 a 16 (in un ordine qualsiasi) e poi li visualizzi in una matrice 4 per 4. La matrice dovrà essere seguita dalla somma delle righe, delle colonne e delle diagonali:

Enter the numbers from 1 to 16 in any order:

16 3 2 13 5 10 11 8 9 6 7 12 4 15 14 1

16 3 2 13

5 10 11 8

9 6 7 12

4 15 14 1

Row sums: 34 34 34 34

Column sums: 34 34 34 34

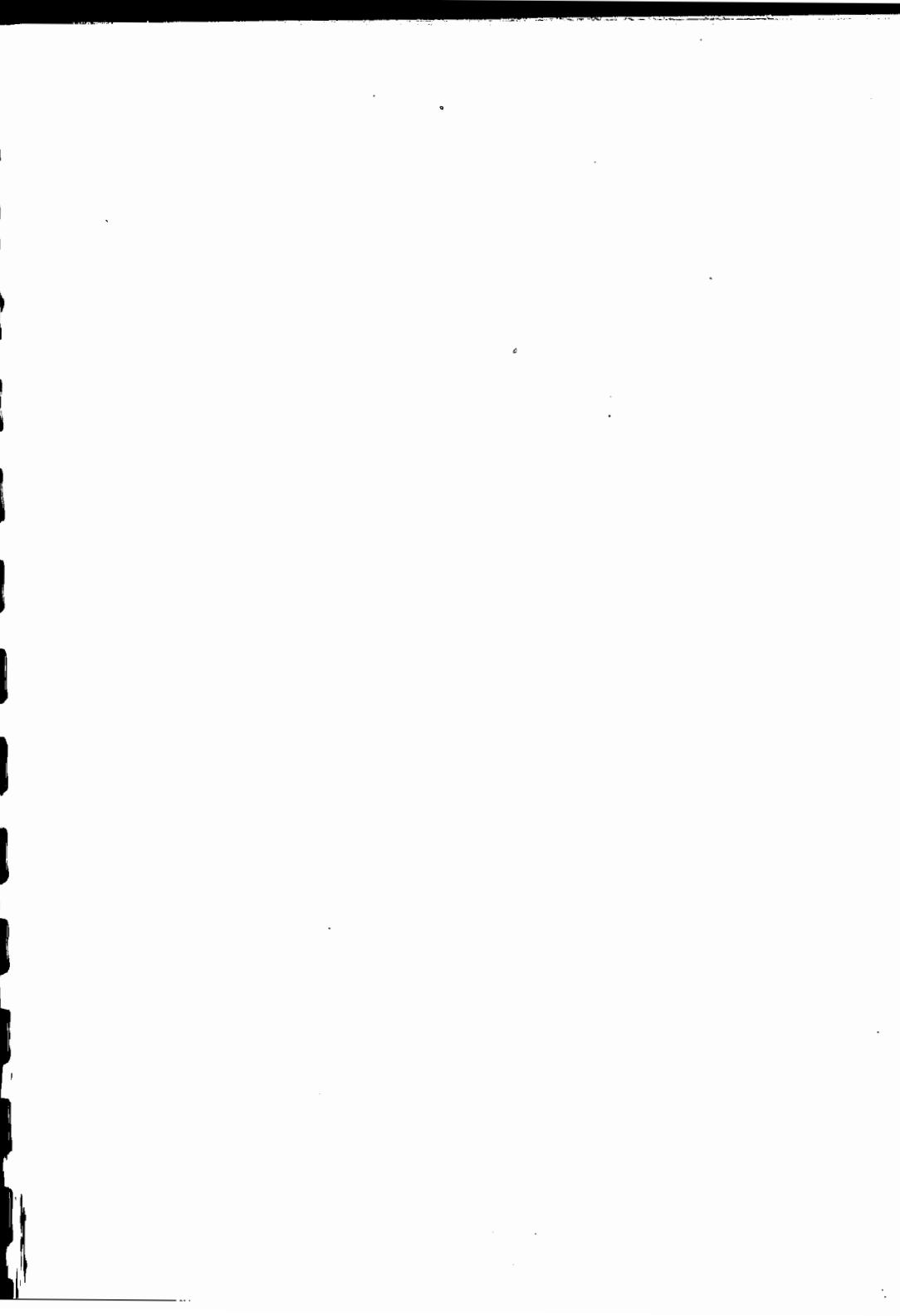
Diagonal sums: 34 34

Se le somme delle righe, delle colonne e delle diagonali sono identiche (come in questo esempio) si dice che i numeri formino il cosiddetto quadrato magico. Il quadrato magico illustrato nell'esempio appare in una incisione del 1514 dell'artista e matematico Albrecht Dürer (osservate che i numeri centrali dell'ultima riga corrispondono alla data dell'incisione).

6. Modificate il programma addfrac.c della Sezione 3.2 in modo che l'utente immetta allo stesso tempo entrambe le frazioni separate da un segno più:

Enter two fractions separated by a plus sign: 5/6+3/4

The sum is 38/24



4 Espressioni

Una delle caratteristiche distintive del C è la sua enfasi sulle espressioni (formule che mostrano come calcolare un valore) piuttosto che sulle istruzioni. Le espressioni più semplici sono le variabili e le costanti. Una variabile rappresenta un valore che deve essere calcolato mentre il programma è in esecuzione, mentre una costante rappresenta un valore che non verrà modificato. Le espressioni più complicate applicano degli operatori sugli operandi (i quali sono a loro volta delle espressioni). Nell'espressione $a+(b*c)$, l'operatore + viene applicato agli operandi a e $(b*c)$, i quali sono a loro volta delle espressioni.

Gli operatori sono gli strumenti base per costruire le espressioni e il C ne possiede una ricca collezione. Per cominciare il C fornisce gli operatori rudimentali che sono presenti in molti linguaggi di programmazione:

- operatori aritmetici, che includono l'addizione, la sottrazione, la moltiplicazione e la divisione;
- operatori relazionali per eseguire confronti come “*i è maggiore di 0*”;
- operatori logici per costruire condizioni come “*i è maggiore di 0 e i è minore di 10*”.

Tuttavia il C non si ferma qui, ma prosegue fornendo dozzine di altri operatori. Agli effetti pratici vi sono talmente tanti operatori che avremo bisogno dei primi venti capitoli del libro per poterli introdurre gradualmente. Padroneggiare così tanti operatori può essere un compito davvero ingrato, tuttavia è essenziale per diventare un valido programmatore C.

In questo capitolo tratteremo alcuni dei più importanti operatori: gli operatori aritmetici (Sezione 4.1), di assegnamento (Sezione 4.2) e di incremento e decremento (Sezione 4.3). La Sezione 4.1, inoltre, illustra la precedenza tra gli operatori e l'associatività, aspetti molto importanti per le espressioni che contengono più di un operatore. La Sezione 4.4 descrive come vengono valutate le espressioni C. Infine, la Sezione 4.5 introduce l'*expression statement*, una caratteristica inusuale che permette di utilizzare una qualsiasi espressione come un'istruzione.

4.1 Operatori aritmetici

Gli **operatori aritmetici** (operatori che eseguono l'addizione, la sottrazione, la moltiplicazione e la divisione) sono i "cavalli da lavoro" di molti linguaggi di programmazione, C incluso. La Tabella 4.1 illustra gli operatori aritmetici del C.

Tabella 4.1 Operatori aritmetici

Unario		Binario	
		Additivo	Moltiplicativo
+ più unario		+ somma	* moltiplicazione
- meno unario		- sottrazione	/ divisione
			% resto

Gli operatori additivi e moltiplicativi vengono detti **binari** perché richiedono *due* operandi. Gli operatori **unari** richiedono un solo operando:

```
i = +1;          /* il + utilizzato come operatore unario */
j = -1;          /* il - utilizzato come operatore unario */
```

l'operatore unario + non fa nulla, infatti non esiste nemmeno nel K&R C. Viene utilizzato solamente per sottolineare che una costante numerica è positiva.

Gli operatori binari probabilmente sono noti. L'unica eccezione potrebbe essere il %, l'operatore resto. Il valore di $i \% j$ è il resto che si ottiene dividendo i per j . Per esempio: $10 \% 3$ è uguale a 1 mentre il valore di $12 \% 4$ è pari a 0.

D&R Gli operatori della Tabella 4.1 (a eccezione di %) ammettono sia operandi interi che a virgola mobile, inoltre è ammesso persino mischiare i tipi. Quando operandi int e float vengono mischiati il risultato è di tipo float. Quindi $9 + 2.5f$ ha valore 11.5 e $6.7f / 2$ ha valore 3.35.

Gli operatori / e % richiedono un'attenzione particolare.

- L'operatore / può produrre risultati inattesi. Quando entrambi gli operandi sono interi l'operatore / "tronca" il risultato omettendo la parte frazionaria. Quindi il valore di $1 / 2$ è 0 e non 0.5.
- L'operatore % richiede operandi interi. Se anche uno solo degli operandi non è un intero, allora il programma non verrà compilato.
- Utilizzare lo zero come operando destro di uno dei due operatori / e % provoca un comportamento non definito [comportamento non definito > 4.4].
- Descrivere il risultato del caso in cui / o % vengono utilizzati con un operando negativo è complesso. Lo standard C89 afferma che se un operando è negativo il risultato della divisione può essere arrotondato sia per eccesso che per difetto (per esempio il valore di $-9 / 7$ può essere sia -1 che -2). Per il C89 se le variabili i o j sono negative allora il segno di $i \% j$ dipende dall'implementazione (per esempio il valore di $-9 \% 7$ può valere sia -2 che 5). D'altra parte per lo standard C99 il risultato di una divisione viene sempre arrotondato verso lo zero (quindi $-9 / 7$ è uguale a -1) e il valore di $i \% j$ ha sempre lo stesso segno di i (di conseguenza $-9 \% 7$ è uguale a -2).

Comportamento definito dall'implementazione

Il termine **definito dall'implementazione** (*implementation-defined*) si presenterà così di frequente nel libro che vale la pena spendere qualche riga per commentarlo. Lo standard C non specifica deliberatamente alcune parti del linguaggio intendendo lasciare all'*implementazione* (il software necessario su una particolare piattaforma per compilare, fare il linking ed eseguire i programmi) il compito di occuparsi dei dettagli. Il risultato è che il comportamento dei programmi può variare in qualche modo da un'implementazione all'altra. Nel C89 il comportamento degli operatori / e % con gli operandi negativi è un esempio di comportamento definito dall'implementazione.

Non specificare parti del linguaggio potrebbe sembrare strano o persino pericoloso ma riflette la filosofia del C. Uno degli obiettivi del linguaggio è l'efficienza, che spesso significa avvicinarsi al comportamento dell'hardware. Alcune CPU restituiscono -1 quando -9 viene diviso per 7 mentre altre restituiscono -2. Lo standard C89 riflette semplicemente questo fatto.

È meglio evitare di scrivere programmi che dipendono dalle caratteristiche definite dall'implementazione. Se questo non è possibile almeno controllate il manuale attentamente (lo standard C richiede che i comportamenti definiti dall'implementazione vengano tutti documentati).

Precedenza degli operatori e associatività

Quando un'espressione contiene più di un operatore allora la sua interpretazione potrebbe non essere immediata. Per esempio $i + j * k$ significa "somma i a j e poi moltiplica il risultato per k" oppure "moltiplica j e k e poi somma il risultato a i"? Una soluzione al problema è quella di aggiungere le parentesi scrivendo $(i + j) * k$ o $i + (j * k)$. Come regola generale il C ammette in tutte le espressioni l'utilizzo di parentesi per effettuare dei raggruppamenti.

Cosa succede se non utilizziamo le parentesi? Il compilatore interpreterà $i + j * k$ come $(i + j) * k$ o come $i + (j * k)$? Come diversi altri linguaggi il C utilizza delle regole di **precedenza degli operatori** per risolvere delle potenziali ambiguità. Gli operatori aritmetici utilizzano il seguente ordine di precedenza:

precedenza più alta: + - (unario)
 * / %

precedenza più bassa: + - (binario)

Gli operatori elencati sulla stessa linea (come + e -) hanno il medesimo **ordine di precedenza**.

Quando due o più operatori appaiono nella stessa espressione possiamo determinare quale sarà l'interpretazione dell'espressione data dal compilatore aggiungendo ripetutamente le parentesi attorno alle sottoespressioni, partendo dagli operatori con maggiore precedenza e proseguendo fino agli operatori con precedenza minore. Gli esempi seguenti illustrano il risultato:

$i + j * k$	è equivalente a	$i + (j * k)$
$-i * -j$	è equivalente a	$(-i) * (-j)$
$+i + j / k$	è equivalente a	$(+i) + (j / k)$

Le regole di precedenza degli operatori non sono sufficienti quando un'espressione contiene due o più operatori dello stesso livello di precedenza. In questa situazione entra in gioco l'associatività degli operatori. Un operatore è detto **associativo** se

a sinistra (*left associative*) se raggruppa gli operandi da sinistra a destra. Gli operatori aritmetici binari (*, /, %, + e -) sono tutti associativi a sinistra e quindi:

$i \circ j - k$ è equivalente a $(i - j) - k$

$i * j / k$ è equivalente a $(i * j) / k$

Un operatore è **associativo a destra** (*right associative*) se raggruppa gli operandi da destra a sinistra. Gli operatori aritmetici unari (+ e -) sono entrambi associativi a destra e quindi

$+ i$ è equivalente a $- (+ i)$

Le regole di precedenza ed associatività sono importanti in molti linguaggi ma lo sono in modo particolare per il C. Il linguaggio C ha così tanti operatori (all'incirca cinquanta!) che pochi programmati si preoccupano di memorizzare le regole di precedenza ed associatività, ma consultano le tabelle degli operatori quando hanno dei dubbi e semplicemente usano molte parentesi [tabelle degli operatori > Appendice A].

Calcolare il carattere di controllo dei codici a barre

Per un certo numero di anni i produttori di beni venduti all'interno degli Stati Uniti e in Canada hanno messo codici a barre su ogni prodotto. Questo codice conosciuto come *Universal Product Code* (UPC) identifica sia il produttore che il prodotto. Ogni codice a barre rappresenta un numero a dodici cifre che viene solitamente stampato sotto le barre. Per esempio il seguente codice a barre viene da un involucro di *Stouffer French Bread Pepperoni Pizza*:



Le cifre

0 13800 15173 5

Appaiono sotto il codice a barre. La prima cifra identifica la tipologia di prodotto (0 o 7 per la maggior parte dei prodotti, 2 per i prodotti che devono essere pesati, 3 per i farmaci e i prodotti relativi alla salute e 5 per i buoni sconto). Il primo gruppo di cinque cifre identifica il produttore (13800 è il codice per Nestlé USA's Frozen Food Division). Il secondo gruppo di cinque cifre identifica il prodotto (inclusa le dimensioni dell'involucro). La cifra finale è una "cifra di controllo" il cui unico scopo è quello di identificare errori nelle cifre precedenti. Se il codice UPC non viene letto correttamente, con buona probabilità le prime 11 cifre non saranno coerenti con l'ultima e lo scanner del negozio rifiuterà l'intero codice. Questo è il metodo per calcolare la cifra di controllo:

Sommare la prima, la terza, la quinta, la settima, la nona e la undicesima cifra.

Sommare la seconda, la quarta, la sesta, l'ottava e la decima cifra.

Moltiplicare la prima somma per 3 e sommarla alla seconda somma.

Sottrarre 1 dal totale.

Calcolare il resto del totale diviso per 10.

Sottrarre il resto dal numero 9.

Usando l'esempio di *Stouffer* abbiamo $0 + 3 + 0 + 1 + 1 + 3 = 8$ per la prima somma e $1 + 8 + 0 + 5 + 7 = 21$ per la seconda somma. Moltiplicando la prima somma per 3 e sommando la seconda rende 45. Sottraendo 1 otteniamo 44. Il resto dalla divisione per 10 è 4. Quando il resto viene sottratto a 9 il risultato è 5. Qui ci sono una coppia di altri codici UPC nel caso voleste esercitarvi nel calcolare la cifra di controllo:

Jif Creamy Peanut Butter (18 oz.): 0 51500 24128 ?

Ocean Spray Jellied Cranberry Sauce (8 oz.): 0 31200 01005 ?

Potete trovare i risultati alla fine della pagina.*

Scriviamo un programma che calcola la cifra di controllo per un qualsiasi codice UPC. Chiederemo all'utente di immettere le prime 11 cifre del codice a barre e successivamente visualizzeremo la corrispondente cifra di controllo. Per evitare confusioni chiederemo all'utente di immettere il numero in tre parti distinte: la cifra singola alla sinistra, il primo gruppo di cinque cifre e il secondo gruppo di cinque cifre. Ecco come dovrebbe apparire una sessione del programma:

Enter the first (single) digit: 0

Enter the first group of five digits: 13800

Enter the second group of five digits: 15173

Check digit: 5

Invece di leggere ogni gruppo come un numero a cinque cifre, lo leggeremo come cinque numeri di una sola cifra. Leggere i numeri come cifre singole è più conveniente e permette di non doverci preoccupare che un numero a cinque cifre possa essere troppo grande per essere memorizzato in una variabile int (alcuni vecchi compilatori limitano il massimo valore di una variabile int a 32767). Per leggere singole cifre utilizzeremo la scanf con la specifica di conversione %d che corrisponde a un intero su singola cifra.

```
upcc /* Calcola la cifra di controllo dei codici a barre */
#include <stdio.h>
int main(void)
{
    int d, i1, i2, i3, i4, i5, j1, j2, j3, j4, j5,
        first_sum, second_sum, total;

    printf("Enter the first (single) digit: ");
    scanf("%d", &d);
    printf("Enter first group of five digits: ");
    scanf("%d%d%d%d%d", &i1, &i2, &i3, &i4, &i5);
```

* Le cifre mancanti sono 8 (Jif) e 6 (Ocean Spray).

```

    printf("Enter second group of five digits: ");
    scanf("%d%d%d%d%d", &j1, &j2, &j3, &j4, &j5);
    first_sum = d + i2 + i4 + j1 + j3 + j5;
    second_sum = i1 + i3 + i5 + j2 + j4;
    total = 3 * first_sum + second_sum;
    printf("Check digit: %d\n", 9 - ((total - 1) % 10));
    return 0;
}

```

Fate caso al fatto che l'espressione $9 - ((\text{total} - 1) \% 10)$ avrebbe potuto essere scritta come $9 - (\text{total} - 1) \% 10$ ma l'insieme aggiuntivo di parentesi la rende molto più comprensibile.

4.2 Operatori di assegnamento

Di solito, una volta che un'espressione è stata calcolata, abbiamo bisogno di memorizzare il suo valore all'interno di una variabile per poterlo utilizzare successivamente. L'operatore = del C (chiamato **assegnamento semplice** o *simple assignment*) viene utilizzato proprio per questo scopo. Per aggiornare il valore già memorizzato all'interno di una variabile, invece, il C fornisce un buon assortimento di operatori di assegnamento secondari.

Assegnamento semplice

L'effetto dell'assegnamento $v = e$ è quello di calcolare l'espressione e e di copiarne il valore all'interno di v . Così come mostrano i seguenti esempi, e può essere una costante, una variabile, oppure un'espressione più complessa:

```

i = 5;           /* adesso i vale 5 */
j = i;           /* adesso j vale 5 */
k = 10 * i + j; /* adesso k vale 55 */

```

Se v ed e non sono dello stesso tipo, allora il valore di e viene convertito nel tipo di v appena viene effettuato l'assegnamento:

```

int i;
float f;

i = 72.99f;      /* adesso i vale 72 */
f = 136;         /* adesso f vale 136.0 */

```

Ritorneremo più avanti sull'argomento delle conversioni di tipo [conversione durante l'assegnamento > 7.4].

In molti linguaggi di programmazione l'assegnamento è una istruzione, nel C invece è un *operatore* proprio come il +. In altre parole, un assegnamento produce un risultato così come lo produrrebbe la somma di due numeri. Il valore dell'assegnamento $v = e$ è esattamente il valore assunto da v dopo l'assegnamento stesso. Quindi il valore di $i = 72.99f$ è 72 (e non 72.99).

Side Effect

Normalmente non ci aspettiamo che gli operatori modifichino i loro operandi dato che in matematica questo non accadde. Scrivere $i + j$ non modifica né i né j ma calcola semplicemente il risultato sommando $i + j$.

La maggior parte degli operatori non modifica i propri operandi, ma non è per tutti così. Diciamo allora che questi operatori hanno degli **effetti collaterali** (*side effect*) in quanto il loro operando va oltre il semplice calcolo di un valore. L'assegnamento semplice è il primo operatore che abbiamo incontrato che possiede un side effect, infatti modifica il suo operando sinistro. Calcolare l'espressione $i = 0$ produce il risultato 0 e, come side effect, assegna il valore 0 a i .

Dato che l'assegnamento è un operatore, se ne possono concatenare assieme diversi:

$i = j = k = 0;$

L'operatore = è associativo a destra e quindi l'espressione è equivalente a

$i = (j = (k = 0));$

L'effetto è quello di assegnare uno 0 in prima istanza a k , successivamente a j e infine a i .



Fate attenzione ai risultati inaspettati che si possono ottenere in un assegnamento concatenato a causa delle conversioni di tipo:

```
int i;
float f;
f = i = 33.3f;
```

a i viene assegnato il valore 33 e successivamente a f viene assegnato il valore 33.0 (e non 33.3 come potreste pensare).

In generale, un assegnamento della forma $v = e$ è ammesso in tutti i casi in cui è ammissibile un valore del tipo v . Nel seguente esempio l'espressione $j = i$ copia i in j , successivamente al nuovo valore di j viene sommato 1 producendo il nuovo valore di k :

```
i = 1;
k = 1 + (j = 1);
k = 10 * i + j;
printf("%d %d %d\n", i, j, k); /* stampa "1 1 2" */
```

Utilizzare gli operatori di assegnamento in questo modo tipicamente non è una buona idea: inglobare gli assegnamenti ("*embedded assignments*") può rendere i programmi difficili da leggere. Questa pratica inoltre può essere fonte di bachi piuttosto subdoli, così come vedremo nella Sezione 4.4.

Lvalue

Molti degli operatori ammettono come loro operandi variabili, costanti o espressioni contenenti altri operatori. L'operatore di assegnamento, invece, richiede un **lvalue** come suo operando sinistro. Un lvalue (si legge *L-value*) rappresenta un oggetto con-

servato nella memoria del computer, non una costante o il risultato di un calcolo. Le variabili sono degli lvalue, mentre espressioni come 10 o $2 * i$ non lo sono. Fino a ora le variabili sono gli unici lvalue che conosciamo ma nei prossimi capitoli ne incontreremo degli altri.

Dato che gli operatori di assegnamento richiedono un lvalue come operando sinistro, non è possibile mettere altri tipi di espressioni nel lato sinistro degli assegnamenti:

```
12 = i;           /*** SBAGLIATO ***/
i + j = 0;        /*** SBAGLIATO ***/
-1 = j;          /*** SBAGLIATO **/
```

Il compilatore individuerà errori di questo tipo e voi otterrete un messaggio come *invalid lvalue in assignment*.

Assegnamento composto

Gli assegnamenti che utilizzano il vecchio valore di una variabile per calcolare quello nuovo sono molto comuni nei programmi C. La seguente istruzione, per esempio, somma 2 al valore memorizzato in *i*:

```
i = i + 2;
```

Gli operatori di **assegnamento composto** (*compound assignment*) del C ci permettono di abbreviare istruzioni come questa e altre simili. Utilizzando l'operatore `+=` serviranno semplicemente:

```
i += 2; /* è lo stesso di i = i + 2; */
```

L'operatore `+=` somma il valore dell'operando destro alla variabile alla sua sinistra.

Ci sono nove altri operatori composti di assegnamento, inclusi i seguenti:

```
= *= /= %=
```

(Tratteremo i restanti operatori di assegnamento composto in un successivo capitolo [**altri operatori di assegnamento > 20.1**.]) Tutti gli operatori di assegnamento composto lavorano praticamente allo stesso modo:

$v += e$ somma v a e , memorizza il risultato in v

$v -= e$ sottrae e da v , memorizza il risultato in v

$v *= e$ moltiplica v per e , memorizza il risultato in v

$v /= e$ divide v per e , memorizza il risultato in v

$v %= e$ calcola il resto della divisione di v per e , memorizza il risultato in v

Osservate che non abbiamo detto che $v += e$ è “equivalente” a $v = v + e$. Uno dei problemi è la precedenza degli operatori: $i *= j + k$ non è la stessa cosa di $i = i * j + k$. Vi sono anche rari casi in cui $v += e$ differisce da $v = v + e$ a causa del fatto che lo stesso v abbia dei side effect. Osservazioni simili si applicano agli altri operatori di assegnamento composto.



Quando utilizzate gli operatori di assegnamento composto state attenti a non invertire i due caratteri che compongono l'operatore. Invertire i due caratteri potrebbe condurre

a un'espressione accettabile per il compilatore ma che non ha il significato voluto. Per esempio, se intendete scrivere $i += j$ ma digitate al suo posto $i =+ j$ il programma verrà compilato comunque. Sfortunatamente l'ultima espressione è equivalente a $i = (+j)$ che copia semplicemente il valore di j in i .

Gli operatori di assegnamento composto hanno le stesse proprietà dell'operatore $=$. In particolare sono associativi a destra e quindi l'istruzione

$i += j += k;$
significa
 $i += (j += k);$

4.3 Operatori di incremento e decremento

Due delle più comuni operazioni su una variabile sono l'incremento (sommare 1 alla variabile) e il decremento (sottrarre 1 alla variabile). Ovviamente possiamo effettuare queste operazioni scrivendo

```
i = i + 1;
j = j - 1;
```

Gli operatori di assegnamento composto ci permettono di condensare un poco queste istruzioni:

```
i += 1;
j -= 1;
```



Tuttavia il C permette di abbreviare maggiormente incrementi e decrementi utilizzando gli operatori **++** (**incremento**) e **--** (**decremento**).

A prima vista gli operatori di incremento e decremento sono semplicissimi: **++** somma 1 al suo operando mentre **--** sottrae 1. Sfortunatamente questa semplicità è ingannevole. Gli operatori di incremento e decremento possono essere davvero problematici da utilizzare. Una complicazione è data dal fatto che **++** e **--** possono essere usati sia come operatori **prefissi** (**++i** e **--i** per esempio) o come operatori **suffissi** (**i++** e **i--**). La correttezza del programma potrebbe dipendere dall'utilizzo della versione giusta.

Un'altra complicazione è dovuta al fatto che, come gli operatori di assegnamento, anche **++** e **--** possiedono dei side effect, ovvero modificano il valore dei loro operandi. Calcolare il valore dell'espressione **++i** (un "pre-incremento") restituisce $i + 1$ e, come side effect, incrementa i :

```
i = 1;
printf("i vale %d\n", ++i);      /* stampa "i vale 2" */
printf("i vale %d\n", i); /* stampa "i vale 2" */
```

Calcolare l'espressione **i++** (un "post-incremento") produce il risultato i , ma causa anche il successivo incremento di i :

```
i = 1;
printf("i vale %d\n", i++);      /* stampa "i vale 1" */
printf("i vale %d\n", i); /* stampa "i vale 2" */
```

D&R

La prima printf visualizza il valore originale di i prima che questo venga incrementato. La seconda printf stampa il nuovo valore. Come illustrano questi nuovi esempi, `++i` significa "incrementa i immediatamente", mentre `i++` significa "per ora utilizza il vecchio valore di i, ma più tardi incrementalo". Quanto più tardi? Lo standard C non specifica un momento preciso, ma è corretto assumere che la variabile i verrà incrementata prima che venga eseguita l'istruzione successiva.

L'operatore `--` ha proprietà simili:

```
i = 1;
printf("i vale %d\n", --i);      /* stampa "i vale 0" */
printf("i vale %d\n", i); /* stampa "i vale 0" */
i = 1;
printf("i vale %d\n", i--);      /* stampa "i vale 1" */
printf("i vale %d\n", i); /* stampa "i vale 0" */
```

Quando `++` o `--` vengono usati più di una volta all'interno della stessa espressione, il risultato può essere difficile da comprendere. Considerate le seguenti istruzioni:

```
i = 1;
j = 2;
k = ++i + j++;
```

Quali sono i valori di i, j e k a esecuzione terminata? Considerato che i viene incrementata prima che il suo valore venga utilizzato e che j viene incrementata dopo il suo utilizzo, l'ultima istruzione equivale a

```
i = i + 1;
```

```
k = i + j;
j = j + 1;
```

quindi i valori finali di i, j e k sono rispettivamente 2, 3 e 4. Per contro eseguire le istruzioni

```
i = 1;
j = 2;
k = i++ + j++;
```

darà a i, j e k rispettivamente i valori 2, 3 e 3.

Le versioni a suffisso di `++` e `--` hanno precedenza più alta rispetto al più e al meno unari e sono associativi a sinistra. Le versioni a prefisso hanno la stessa precedenza del più e del meno unari e sono associativi a destra.

4.4 Calcolo delle espressioni

La Tabella 4.2 riassume gli operatori che abbiamo visto finora (l'Appendice A ha una tabella simile che illustra *tutti* gli operatori). La prima colonna indica la precedenza relativa di ogni operatore rispetto agli altri presenti nella tabella (la precedenza più alta è 1, la più bassa è 5). L'ultima colonna indica l'associatività di ogni operatore.

La Tabella 4.2 (o la sua versione più estesa nell'Appendice A) ha diversi utilizzi. Soffermiamoci su uno di questi. Supponiamo, durante la lettura di un programma, di imbatterci in una espressione complessa come

```
a = b += c++ - d + --e / -f
```

Tabella 4.2 Un elenco parziale degli operatori C

Precedenza	Nome	Simbolo	Associatività
1	incremento (suffisso)	++	sinistra
	decremento (suffisso)	--	
2	incremento (prefisso)	++	destra
	decremento (prefisso)	--	
	più unario	+	
	meno unario	-	
3	moltiplicativi	* / %	sinistra
4	additivi	+ -	sinistra
5	assegnamento	= *= /= %= += -=	destra

Questa espressione sarebbe stata facile da comprendere se fossero state inserite delle parentesi per rimarcare la sua composizione a partire dalle sottoespressioni. Con l'aiuto della Tabella 4.2 aggiungere le parentesi all'espressione diventa semplice. Dopo aver esaminato l'espressione alla ricerca dell'operatore con precedenza più alta, mettiamo delle parentesi attorno a quest'ultimo e ai suoi operandi. In questo modo indichiamo che da quel punto in avanti il contenuto delle parentesi appena inserite deve essere trattato come un singolo operando. Ripetiamo il procedimento fino a che l'espressione non è stata completamente racchiusa da parentesi.

Nel nostro esempio l'operatore con la precedenza più alta è il ++, utilizzato come operatore suffisso. Racchiudiamo tra le parentesi ++ e il suo operando:

$a = b += (c++) - d + --e / -f$

Ora individuiamo all'interno dell'espressione l'operatore -- e l'operatore meno di tipo unario (entrambi con precedenza 2):

$a = b += (c++) - d + (--e) / (-f)$

Notate che l'altro segno meno ha un operando alla sua immediata sinistra e quindi deve essere considerato come un operatore di sottrazione e non come un operatore meno di tipo unario.

Adesso è la volta dell'operatore / (precedenza 3):

$a = b += (c++) - d + ((--e) / (-f))$

L'espressione contiene due operatori con precedenza 4, la sottrazione e l'addizione. Ogni volta che due operatori con la stessa precedenza sono adiacenti a un operando dobbiamo fare attenzione all'associatività. Nel nostro esempio - e + sono entrambi adiacenti a d e perciò applichiamo le regole di associatività. Gli operatori - e + si gruppano da sinistra a destra e quindi le parentesi vanno inserite prima attorno alla sottrazione e successivamente attorno all'addizione:

$a = b += (((c++) - d) + ((--e) / (-f)))$

Gli unici rimasti sono gli operatori `=` e `+=`. Entrambi sono adiacenti a `b` e quindi si deve tenere conto dell'associatività. Gli operatori di assegnamento raggruppano da destra a sinistra, perciò le parentesi vanno messe prima attorno all'espressione con `+=` e poi attorno all'espressione contenente l'operatore `=`:

```
(a = (b += (((c++) - d) + ((--e) / (-f)))))
```

Ora l'espressione è racchiusa completamente tra le parentesi.

Ordine nel calcolo delle sottoespressioni

Le regole di precedenza e associatività degli operatori ci permettono di suddividere qualsiasi espressione C in sottoespressioni (in tal modo si determina in modo univoco la posizione delle parentesi). Paradossalmente queste regole non ci permettono di determinare sempre il valore dell'espressione, infatti questo può dipendere dall'ordine in cui le sottoespressioni vengono calcolate.

Il C non stabilisce l'ordine in cui le sottoespressioni debbano essere calcolate (con l'eccezione delle sottoespressioni contenenti l'`and` logico, l'`or` logico, l'operatore condizionale e l'operatore virgola [operatori logici and e or > 5.1; operatore condizionale > 5.2; operatore virgola > 6.3]. Quindi nell'espressione `(a + b) * (c - d)` non sappiamo se `(a + b)` verrà calcolata prima di `(c - d)`.

La maggior parte delle espressioni hanno lo stesso valore indipendentemente dall'ordine con cui le loro sottoespressioni vengono calcolate. Tuttavia questo potrebbe non essere vero nel caso in cui una sottoespressione modifichasse uno dei suoi operandi. Considerate l'esempio seguente:

```
a = 5;  
c = (b = a + 2) - (a = 1);
```

L'effetto dell'esecuzione della seconda espressione non è definito, lo standard del C non spiega che cosa dovrebbe verificarsi. Con molti compilatori il valore di `c` potrebbe essere sia 6 che 2. Se la sottoespressione `(b = a + 2)` viene calcolata per prima, allora `a b` viene assegnato il valore 7 e `a c` il valore 6. Tuttavia se `(a + 1)` viene calcolata per prima, allora `a b` viene assegnato il valore 3 e `a c` il valore 2.



Evitate le espressioni che in alcuni punti accedono al valore di una variabile e in altri lo modificano. L'espressione `(b = a + 2) - (a = 1)` accede al valore di `a` (in modo da calcolare `a + 2`) e ne modifica anche il valore (assegnando 1 ad `a`). Quando incontrano espressioni del genere, alcuni compilatori potrebbero produrre un messaggio di warning come `operation on 'a' may be undefined`.

Per scongiurare problemi è buona pratica evitare l'uso di operatori di assegnamento all'interno delle sottoespressioni. Piuttosto conviene utilizzare una serie di assegnamenti separati; per esempio, l'istruzione appena incontrata potrebbe essere scritta come

```
a = 5;  
b = a + 2;  
a = 1;  
c = b - a;
```

A esecuzione terminata il valore di c sarà sempre 6.

Oltre agli operatori di assegnamento, gli unici che modificano i loro operandi sono quelli di incremento e di decremento. Quando utilizzate questi operatori fate attenzione affinché la vostra espressione non dipenda da un particolare ordine di calcolo. Nell'esempio seguente a j può venir assegnato uno qualsiasi tra due valori:

```
i = 2;
j = i * i++;
```

Appare naturale assumere che a j venga assegnato il valore 4. Tuttavia, l'effetto legato all'esecuzione dell'istruzione non è definito e a j potrebbe benissimo venir assegnato il valore 6. La situazione è questa: (1) il secondo operando (il valore originario di i) viene *caricato* e successivamente la variabile i viene incrementata. (2) Il primo operando (il nuovo valore di i) viene *caricato*. (3) Il nuovo e il vecchio valore di i vengono moltiplicati tra loro ottenendo 6. "Caricare" una variabile significa recuperare dalla memoria il valore della variabile stessa. Un cambiamento successivo a tale valore non avrebbe effetto sul valore caricato, il quale viene tipicamente memorizzato all'interno della CPU in una speciale locazione (conosciuta come **registro** [registri > 18.2]).

Comportamento indefinito

In maniera conforme allo standard C le istruzioni $c = (b = a + 2) - (a = 1)$; e $j = i * i++$; causano un comportamento indefinito che è una cosa differente rispetto al comportamento definito dall'implementazione (si veda la Sezione 4.1). Quando un programma si avventura nel regno del comportamento indefinito non si possono fare previsioni. Il programma potrà assumere comportamenti differenti a seconda del compilatore utilizzato. Tuttavia questa non è l'unica cosa che potrebbe accadere. In primo luogo il programma potrebbe non essere compilabile, se venisse compilato potrebbe non essere eseguibile, e nel caso in cui fosse eseguibile potrebbe andare in crash, comportarsi in modo erratico o produrre risultati senza senso. In altre parole i comportamenti indefiniti devono essere evitati come la peste.

4.5 Expression statement

Il C possiede un'insolita regola secondo la quale qualsiasi espressione può essere utilizzata come un'istruzione. Quindi, qualunque espressione (indipendentemente dal suo tipo e da cosa venga calcolato) può essere trasformata in una istruzione aggiungendo un punto e virgola. Per esempio, possiamo trasformare l'espressione $++i$ nell'istruzione:

```
++i;
```

Quando questa istruzione viene eseguita, per prima cosa i viene incrementata e poi viene caricato il nuovo valore di i (così come se dovesse essere utilizzata in un'espressione che racchiude la prima). Tuttavia, dato che $++i$ non fa parte di un'espressione più grande, il suo valore viene scartato e viene eseguita l'istruzione successiva (ovviamente la modifica di i è permanente).

Considerando che il suo valore viene scartato, non c'è motivo di utilizzare un'espressione come se fosse una istruzione a meno che l'espressione non abbia un *side effect*. Diamo un'occhiata a tre esempi. Nel primo, i viene memorizzato in i e in seguito il nuovo valore di i viene caricato ma non usato:

i = 1;

Nel secondo esempio il valore di i è caricato ma non utilizzato, tuttavia la variabile i viene decrementata in un secondo momento:

i--;

Nel terzo esempio il valore dell'espressione i * j - 1 viene calcolato e successivamente scartato:

i * j - 1;

Dato che i e j non vengono modificati questa istruzione non ha alcun effetto e quindi è inutile.



Un dito che scivola sulla tastiera potrebbe creare facilmente un'espressione che non fa nulla. Per esempio, invece di scrivere

i = j;

potremmo digitare accidentalmente

i + j;

(Questo tipo di errori è comune se si utilizza una tastiera americana, perché i caratteri = e + occupano solitamente lo stesso tasto [N.d.T.]) Alcuni compilatori possono rilevare degli *expression statement* senza significato restituendo un messaggio di warning come *statement with no effect*.

Domande e risposte

D: Abbiamo notato che il C non ha un operatore esponenziale. Come posso elevare a potenza un numero?

R: Il miglior modo per elevare un numero intero per una piccola potenza intera è quello delle moltiplicazioni successive ($i * i * i$ è i elevato al cubo). Per elevare un numero a una potenza non intera chiamate la funzione pow [**pow function > 23.3**].

D: Vogliamo applicare l'operatore % a un operando a virgola mobile, ma il nostro programma non compila. Come possiamo fare? [p. 56]

R: L'operatore % richiede degli operandi interi. Utilizzate al suo posto la funzione fmod [**fmod > 23.3**].

D: Perché le regole per l'utilizzo degli operatori / e % con operandi negativi sono così complicate? [p. 56]

R: Le regole non sono così complicate come potrebbe apparire. Sia nel C89 che nel C99 l'obiettivo è quello di assicurarsi che il valore di $(a / b) * b + a \% b$ sia sempre uguale ad a (e infatti entrambi gli standard garantiscono che questo avvenga nel caso

in cui il valore di a / b sia "rappresentabile"). Il problema è che per a / b e $a \% b$ ci sono due modi di soddisfare questa equazione nei casi in cui a o b sono negativi. Come abbiamo già visto, nel C89, possiamo avere che $-9 / 7$ sia uguale a -1 e che $-9 \% 7$ valga -2 , oppure che $-9 / 7$ sia uguale a -2 e che $-9 \% 7$ valga 5 . Nel primo caso, $(-9 / 7) * 7 + -9 \% 7$ ha valore $-1 \times 7 + -2 = -9$. Nel secondo caso $(-9 / 7) * 7 + -9 \% 7$ ha valore $-2 \times 7 + 5 = -9$. Al momento in cui il C99 ha iniziato a circolare, la maggior parte delle CPU erano progettate per troncare verso lo zero il risultato della divisione e così questo comportamento è stato inserito all'interno dello standard come l'unico ammesso.

D: Se il C ha gli lvalue ha anche gli rvalue? [p.61]

R: Sì, certamente. Un *lvalue* è un'espressione che può apparire sul lato sinistro di un assegnamento, mentre un *rvalue* è un'espressione che può apparire sul lato destro. Quindi un *rvalue* può essere una variabile, una costante o un'espressione più complicata. In questo libro, come nel C standard, utilizzeremo il termine "espressione" invece che "rvalue".

D*: Abbiamo detto che $v += e$ non è equivalente a $v = v + e$ nel caso in cui v abbia un side effect. Potrebbe spiegare meglio? [p. 62]

R: Calcolare $v += e$ fa in modo che v venga valutata un volta sola. Calcolare $v = v + e$ fa in modo che v venga valutata due volte. Nel secondo caso un qualsiasi *side effect* causato dal calcolo di v si verificherà due volte. Nel seguente esempio i viene incrementato una volta:

```
a[i++] += 2;
```

Ecco come si presenterà l'istruzione utilizzando un $=$ al posto del $+=$:

```
a[i++] = a[i++] + 2;
```

Il valore di i viene modificato così come accadrebbe se fosse usato in altre parti dell'istruzione e quindi l'effetto di tale istruzione non è definito. È probabile che i venga incrementato due volte, tuttavia non possiamo affermare con certezza quello che accadrà.

D: Perché il C fornisce gli operatori $++$ e $--$? Sono più veloci degli altri sistemi per incrementare o decrementare oppure sono solamente più comodi? [p. 63]

R: Il C ha ereditato $++$ e $--$ dal precedente linguaggio di Ken Thompson, il B. Apparentemente Thompson creò questi operatori perché il suo compilatore B era in grado di generare una traduzione più compatta per $++i$ rispetto a quella per $i = i + 1$. Questi operatori sono diventati una parte integrante del C (infatti la maggior parte degli idiom C si basano su essi). Con i compilatori moderni, tuttavia, utilizzare $++$ e $--$ non renderà il programma né più piccolo né più veloce. La costante popolarità di questi operatori deriva principalmente dalla loro brevità e comodità di utilizzo.

D: Gli operatori $++$ e $--$ funzionano con le variabili float?

R: Sì, le operazioni di incremento e decremento possono essere applicate ai numeri a virgola mobile nello stesso modo in cui possono essere applicate agli interi. Tuttavia nella pratica è piuttosto raro incrementare o decrementare una variabile float.

D*: Quando vengono eseguiti esattamente l'incremento o il decremento nei casi in cui si utilizzano le versioni a suffisso di `++` e `--`? [p. 64]

R: Questa è un'ottima domanda. Sfortunatamente la risposta è piuttosto complessa. Lo standard C introduce il concetto di *sequence point* e dice che "l'aggiornamento del valore conservato di un operando deve avvenire tra il *sequence point* precedente e il successivo". Ci sono diversi tipi di *sequence point* in C, la fine di un *expression statement* ne è un esempio. Alla fine di un *expression statement* tutti i decrementi e gli incrementi presenti all'interno dell'istruzione devono essere eseguiti. L'istruzione successiva non può essere eseguita fino a che questa condizione non viene rispettata.

Alcuni operatori, che incontreremo nei prossimi capitoli (l'*and* logico, l'*or* logico, l'operatore condizionale e la virgola), impongono a loro volta dei *sequence point*. Lo stesso fanno anche le chiamate a funzione: gli argomenti in una chiamata a funzione devono essere calcolati prima che la chiamata possa essere eseguita. Se capita che un argomento faccia parte di un'espressione contenente un operatore `++` o un operatore `--`, allora l'incremento e il decremento devono essere eseguiti prima che la chiamata a funzione abbia luogo.

D: Cosa intendeva quando ha detto che il valore di un *expression statement* viene scartato?

R: Per definizione un'espressione rappresenta un valore. Se per esempio `i` possiede il valore 5 allora il calcolo di `i + 1` produce il valore 6. Trasformiamo `i + 1` in un'istruzione ponendo un punto e virgola dopo l'espressione:

`i + 1;`

Quando questa istruzione viene eseguita il valore `i + 1` viene calcolato. Dato che non abbiamo salvato questo valore in una variabile (e nemmeno lo abbiamo utilizzato in altro modo) allora questo viene perso.

D: Ma cosa succede con istruzioni tipo `i = 1;`? Non capiamo cosa venga scartato.

R: Non dimenticate che in C l'operatore `=` produce un valore così come ogni altro operatore. L'assegnamento

`i = 1;`

assegna 1 a `i`. Il valore dell'intera espressione è 1 che viene scartato. Scartare il valore dell'espressione non è una perdita grave visto che la ragione per la scrittura di questa istruzione è in primo luogo quella di modificare `i`.

Esercizi

Esercizio 4.1

1. Mostrate l'output di ognuno dei seguenti frammenti di programma. Assumete che `i`, `j` e `k` siano variabili `int`.

- (a) `i = 5; j = 3;`
`printf("%d %d", i / j, i % j);`
- (b) `i = 2; j = 3;`
`printf("%d", (i + 10) % j);`

(c) $i = 7; j = 8; k = 9;$
 printf("%d", (i + 10) % k / j);
(d) $i = 1; j = 2; k = 3;$
 printf("%d", (i + 5) % (j + 2) / k);

- W 2. * Se i e j sono interi positivi, $(-i)/j$ ha sempre lo stesso valore di $-(i/j)$? Motivate la risposta.
3. Qual è il valore di ognuna di queste espressioni nello standard C89? (Fornite tutti i possibili valori se un'espressione può averne più di uno).
- (a) $8 / 5$
(b) $-8 / 5$
(c) $8 / -5$
(d) $-8 / -5$
4. Ripetete l'Esercizio 3 per il C99.
5. Qual è il valore di ognuna delle seguenti espressioni nello standard C89? (Fornite tutti i possibili valori se un'espressione può averne più di uno).
- (a) $8 \% 5$
(b) $-8 \% 5$
(c) $8 \% -5$
(d) $-8 \% -5$
6. Ripetete l'Esercizio 5 per il C99.
7. L'algoritmo per calcolare la cifra di controllo dei codici a barre termina con i seguenti passi:

Sottrarre 1 dal totale.

Calcolare il resto ottenuto dividendo per 10 il totale riaggiustato.

Sottrarre il resto da 9.

Si cerchi di semplificare l'algoritmo utilizzando al loro posto questi passi:

Calcolare il resto ottenuto dividendo per 10 il totale.

Sottrarre il resto da 10.

Perché questa tecnica non funziona?

8. Il programma `upc.c` funzionerebbe ugualmente se l'espressione $9 - ((total - 1) \% 10)$ fosse rimpiazzata dall'espressione $(10 - (total \% 10)) \% 10$?

- Sezione 4.2 W 9. Mostrate l'output di ognuno dei seguenti frammenti di programma. Assumete che i , j e k siano variabili int.

(a) $i = 7; j = 8;$
 $i *= j + 1;$
 printf("%d %d", i, j);
(b) $i = j = k = 1;$
 $i += j += k;$
 printf("%d %d %d", i, j, k);
(c) $i = 1; j = 2; k = 3;$
 $i -= j -= k;$
 printf("%d %d %d", i, j, k);

```
(d) i = 2; j = 1; k = 0;
    i *= j *= k;
    printf("%d %d %d", i, j, k);
```

10. Mostrate l'output di ognuno dei seguenti frammenti di programma. Assumete che i, j e k siano variabili int.

```
(a) i = 6;
    j = i += i;
    printf("%d %d", i, j);
(b) i = 5;
    j = (i -= 2) + 1;
    printf("%d %d", i, j);
(c) i = 7;
    j = 6 + (i = 2.5);
    printf("%d %d", i, j);
(d) i = 2; j = 8;
    j = (i = 6) + (j = 3);
    printf("%d %d", i, j);
```

- Sezione 4.3 11. Mostrate l'output di ognuno dei seguenti frammenti di programma. Assumete che i, j e k siano variabili int.

```
(a) i = 1;
    printf("%d ", i++ - 1);
    printf("%d", i);
(b) i = 10; j = 5;
    printf("%d ", i++ - ++j);
    printf("%d %d", i, j);
(c) i = 7; j = 8;
    printf("%d ", i++ - --j);
    printf("%d %d", i, j);
(d) i = 3; j = 4; k = 5;
    printf("%d ", i++ - j++ + --k);
    printf("%d %d %d", i, j);
```

12. Mostrate l'output di ognuno dei seguenti frammenti di programma. Assumete che i, j e k siano variabili int.

```
(a) i = 5;
    j = ++i * 3 - 2;
    printf("%d %d", i, j);
(b) i = 5;
    j = 3 - 2 * i++;
    printf("%d %d", i, j);
(c) i = 7;
    j = 3 * i-- + 2;
    printf("%d %d", i, j);
(d) i = 7;
    j = 3 + --i * 2;
    printf("%d %d", i, j);
```

- W 13. Quale delle due espressioni `++i` e `i++` equivale a `(i += 1)`? Motivate la vostra risposta.

Sezione 4.4 14. Introducete le parentesi per indicare come ognuna delle seguenti espressioni verrebbe interpretata da un compilatore C.

- (a) `a * b - c * d + e`
- (b) `a / b % c / d`
- (c) `- a - b + c - + d`
- (d) `a * - b / c - d`

Sezione 4.5 15. Fornite i valori assunti da `i` e `j` dopo l'esecuzione di ciascuno dei seguenti *expression statement* (Assumete che inizialmente `i` abbia il valore 1 e `j` il valore 2).

Progetti di programmazione

1. Scrivete un programma che chieda all'utente di immettere un numero a due cifre e successivamente stampi il numero con le cifre invertite. Una sessione del programma deve presentarsi come segue:

Enter a two-digit number: 28

The reversal is: 82

Leggete il numero usando la specifica `%d` e poi suddividetelo in due cifre. Suggerimento: Se `n` è un intero allora `n%10` è l'ultima cifra di `n` mentre `n/10` è `n` con l'ultima cifra rimossa.

- W 2. Estendete il programma del Progetto di Programmazione 1 per gestire numeri a tre cifre.
3. Riscrivete il programma del Progetto di Programmazione 2 in modo che stampi la scrittura inversa di un numero a tre cifre senza utilizzare calcoli aritmetici per dividere il numero in cifre. Suggerimento: Guardate il programma `upc.c` della Sezione 4.1.
 4. Scrivete un programma che legga un numero intero immesso dall'utente e lo visualizzi in base ottale (base 8):

Enter a number between 0 and 32767: 1953

In octal, your number is: 03641

L'output dovrebbe essere visualizzato utilizzando cinque cifre anche nel caso in cui ne fossero sufficienti meno. Suggerimento: Per convertire il numero in ottale dividetelo inizialmente per 8, il resto è l'ultima cifra del numero ottale (1 in questo caso). Dividete ancora il numero originale per 8 prendendo il resto dalla divisione per ottenere la penultima cifra (come vedremo nel Capitolo 7 la `printf` è in grado di stampare numeri in base 8, quindi nella pratica c'è un modo più semplice per scrivere questo programma).

5. Riscrivete il programma `upc.c` della Sezione 4.1 in modo che l'utente immetta 11 cifre in una volta sola invece che immettere il codice in gruppi da una a cinque cifre.

Enter the first 11 digits of a UPC: 01380015173

Check digit: 5

6. I Paesi europei utilizzano un codice a 13 cifre chiamato *European Article Number (EAN)* al posto delle 12 cifre dell'*Universal Product Code (UPC)* utilizzato in Nord America. Ogni EAN termina con una cifra di controllo esattamente come succede per i codici UPC. La tecnica per calcolare il codice di controllo è simile:

Sommare la seconda, la quarta, la sesta, l'ottava, la decima e la dodicesima cifra.

Sommare la prima, la terza, la quinta, la settima, la nona e l'undicesima cifra.

Moltiplicare la prima somma per 3 e sommarla alla seconda somma.

Sottrarre 1 dal totale.

Calcolare il resto ottenuto quando il totale modificato viene diviso per 10.

Sottrarre da 9 il resto.

Per esempio considerate il prodotto *Güllioglu Turkish Delight Pistachio & Coconut* che possiede un codice EAN pari a 8691484260008. La prima somma è $6 + 1 + 8 + 2 + 0 + 0 = 17$, e la seconda somma è $8 + 9 + 4 + 4 + 6 + 0 = 31$. Moltiplicando la prima somma per 3 e sommandole la seconda somma si ottiene 82. Sottraendo 1 si ottiene 81. Il resto della divisione per 10 è 1. Quando il resto viene sottratto da 9 il risultato è 8 che combacia con l'ultima cifra del codice originale. Il vostro compito è quello di modificare il programma *upc.c* della Sezione 4.1 in modo da calcolare la cifra di controllo di un codice EAN. L'utente immetterà le prime 12 cifre del codice EAN come un singolo numero:

Enter the first 12 digits of an EAN: 869148426000

Check digit: 8

5 Istruzioni di selezione

Sebbene il C abbia molti operatori, in compenso ha relativamente poche istruzioni. Finora ne abbiamo incontrate solamente due: l'istruzione `return` [[istruzione return > 2.2](#)] e gli expression statement [[expression statement > 4.5](#)]. La maggior parte delle istruzioni rimanenti ricadono all'interno di tre categorie, a seconda di come influiscono sull'ordine di esecuzione delle istruzioni.

- **Istruzioni di selezione.** Le istruzioni `if` e `switch` permettono al programma di selezionare un particolare percorso di esecuzione fra un insieme di alternative.
- **Istruzioni di iterazione.** Le istruzioni `while`, `do` e `for` permettono le iterazioni (i cosiddetti `loop`).
- **Istruzioni di salto.** Le istruzioni `break`, `continue` e `goto` provocano un salto incondizionato in un altro punto del programma (l'istruzione `return` appartiene a questa categoria).

Le uniche istruzioni rimanenti sono l'istruzione composta, che raggruppa diverse istruzioni in una, e l'istruzione vuota, che non esegue alcuna azione.

Questo capitolo tratta le istruzioni di selezione e l'istruzione composta (il Capitolo 6 tratta le istruzioni di iterazione, le istruzioni di salto e l'istruzione vuota). Prima di poter scrivere istruzioni con il costrutto `if` abbiamo bisogno delle espressioni logiche, ovvero di condizioni che l'istruzione `if` possa verificare. La Sezione 5.1 spiega come le istruzioni logiche vengano costruite a partire dagli operatori relazionali (`<`, `<=`, `>` e `>=`), di uguaglianza (`==` e `!=`) e dagli operatori logici (`&&`, `||`, e `!`). La Sezione 5.2 tratta l'istruzione `if`, l'istruzione composta oltre che l'operatore condizionale (`?:`). Questi costrutti sono in grado di verificare una condizione all'interno di un'espressione. La Sezione 5.3 descrive l'istruzione `switch`.

5.1 Espressioni logiche

Diverse istruzioni C, tra cui l'istruzione `if`, devono verificare il valore di un'espressione per capire se è "vera" o "falsa". Per esempio: un'istruzione `if` potrebbe aver bisogno di verificare l'espressione `i < j`, un valore "vero" indicherebbe che `i` è minore di `j`. In molti linguaggi di programmazione, espressioni come `i < j` possiedono uno

speciale tipo di valore detto "Booleano" o "logico". Questo particolare tipo può assumere solamente due valori: *falso* o *vero*. Al contrario, nel linguaggio C un confronto come $i < j$ restituisce un valore intero: 0 (falso) oppure 1 (vero). Tenendo presente questa particolarità andiamo a vedere gli operatori che vengono utilizzati per costruire espressioni logiche.

Operatori relazionali

Gli **operatori relazionali** del C (Tabella 5.1) corrispondono agli operatori matematici $<$, $>$, \leq e \geq a eccezione del fatto che, quando utilizzati in un'espressione, questi restituiscono il valore 0 (falso) o il valore 1 (vero). Per esempio il valore di $10 < 11$ è 1, mentre il valore di $11 < 10$ è 0.

Tabella 5.1 Operatori relazionali

Simbolo	Significato
$<$	minore di
$>$	maggiore di
\leq	minore o uguale a
\geq	maggiore o uguale a

Gli operatori relazionali possono essere utilizzati per confrontare numeri interi e a virgola mobile ma sono ammessi anche operandi appartenenti a tipi diversi. Quindi $1 < 2.5$ ha valore 1 mentre $5.6 < 4$ ha valore 0.

Il grado di precedenza degli operatori relazionali è inferiore a quello degli altri operatori aritmetici, per esempio $i + j < k - 1$ significa $(i + j) < (k - 1)$. Gli operatori relazionali inoltre sono associativi a sinistra.



L'espressione

$i < j < k$

è ammessa in C, tuttavia non ha il significato che vi potreste aspettare. Dato che l'operatore $<$ è associativo a sinistra questa espressione è equivalente a

$(i < j) < k$

In altre parole questa espressione per prima cosa controlla se i è minore di j , successivamente l'1 o lo 0 prodotto da questo confronto viene confrontato con k . L'espressione *non* controlla se j è compreso tra i e k (vedremo più avanti in questa sezione che l'espressione corretta sarebbe $i < j \&& j < k$).

Operatori di uguaglianza

Nonostante gli operatori relazionali vengano indicati nel C con gli stessi simboli utilizzati in molti altri linguaggi di programmazione, gli *operatori di uguaglianza* sono

contraddistinti da un aspetto particolare (Tabella 5.2). L'operatore di "uguale a" è formato da due caratteri = adiacenti e non da uno solo perché il carattere = preso singolarmente rappresenta l'operatore di assegnazione. Anche l'operatore di "diverso da" viene scritto con due caratteri: !=.

Tabella 5.2 Operatori di uguaglianza

Simbolo	Significato
==	uguale a
!=	diverso da

Così come gli operatori relazionali anche gli operatori di uguaglianza sono associativi a sinistra e producono come risultato uno 0 (falso) oppure un 1 (vero). Tuttavia gli operatori di uguaglianza hanno un ordine di precedenza *inferiore* a quello degli operatori relazionali. Per esempio, l'espressione

$i < j == j < k$

è equivalente a

$(i < j) == (j < k)$

che è vera se le espressioni $i < j$ e $j < k$ sono entrambe vere oppure entrambe false.

I programmatore più abili a volte sfruttano il fatto che gli operatori relazionali e quelli di uguaglianza restituiscano valori interi. Per esempio il valore dell'espressione $(i >= j) + (i == j)$ può essere 0, 1 o 2 a seconda che i sia rispettivamente minore, maggiore o uguale a j. Tuttavia trucchi di programmazione come questo non sono generalmente una buona pratica dato che rendono i programmi più difficili da comprendere.

Operatori logici

Espressioni logiche più complicate possono venir costruite a partire da quelle più semplici grazie all'uso degli **operatori logici**.

Tabella 5.3 Operatori logici

Simbolo	Significato
!	negazione logica
&&	and logico
	or logico

Gli operatori logici producono 0 oppure 1 come loro risultato. Di solito gli operatori logici avranno i valori 0 o 1. Tuttavia questo non è obbligatorio: gli operatori logici trattano un qualsiasi valore diverso da zero come vero e qualsiasi valore uguale a zero come falso.

Gli operatori logici operano in questo modo:

- !expr1 ha il valore 1 se expr1 ha il valore 0.
- $\text{expr1} \&\& \text{expr2}$ ha valore 1 se i valori di expr1 ed expr2 sono entrambi diversi da zero.
- $\text{expr1} \mid\mid \text{expr2}$ ha valore 1 se il valore di expr1 o quello di expr2 (o entrambi) sono diversi da zero.

In tutti gli altri casi questi operatori producono il valore 0.

Sia $\&\&$ che $\mid\mid$ eseguono la "corto circuitazione" del calcolo dei loro operandi. Questo significa che questi operatori per prima cosa calcolano il valore il loro operando sinistro e successivamente quello destro. Se il valore dell'espressione può essere dedotto dal valore del solo operando sinistro, allora l'operatore destro non è viene esaminato. Considerate la seguente espressione:

$(\text{i} != 0) \&\& (\text{j} / \text{i} > 0)$

Per trovare il valore dell'espressione dobbiamo per prima cosa calcolare il valore di $(\text{i} != 0)$. Se i non è uguale a 0, allora abbiamo bisogno di calcolare il valore di $(\text{j} / \text{i} > 0)$ per sapere se l'intera espressione è vera o falsa. Tuttavia se i è uguale a 0 allora l'intera espressione deve essere falsa e quindi non c'è bisogno di calcolare $(\text{j} / \text{i} > 0)$. Il vantaggio della corto circuitazione nel calcolo di questa espressione è evidente: senza di essa si sarebbe verificata una divisione per zero.



Fate attenzione agli effetti secondari delle espressioni logiche. Grazie alla proprietà di corto circuitazione degli operatori $\&\&$ e $\mid\mid$, gli effetti secondari degli operandi non sempre hanno luogo. Considerate la seguente espressione:

$\text{i} > 0 \&\& \text{++j} > 0$

Sebbene j venga apparentemente incrementata come side effect del calcolo dell'espressione, questo non avviene in tutti i casi. Se $\text{i} > 0$ è falso allora $\text{++j} > 0$ non viene calcolato e quindi la variabile j non viene incrementata. Il problema può essere risolto cambiando la condizione in $\text{++j} > 0 \&\& \text{i} > 0$ oppure incrementando j separatamente (che sarebbe una pratica migliore).

L'operatore $!$ possiede il medesimo ordine di precedenza degli operatori più e meno unari. L'ordine di precedenza degli operatori $\&\&$ e $\mid\mid$ è inferiore a quello degli operatori relazionali e di uguaglianza. Per esempio: $\text{i} < \text{j} \&\& \text{k} == \text{m}$ significa $(\text{i} < \text{j}) \&\& (\text{k} == \text{m})$. L'operatore $!$ è associativo a destra, mentre gli operatori $\&\&$ e $\mid\mid$ sono associativi a sinistra.

5.2 L'istruzione if

L'istruzione if permette al programma di scegliere tra due alternative sulla base del valore di un'espressione. Nella sua forma più semplice l'istruzione if ha la struttura:

if (*espressione*) *istruzione*

Tenete presente che le parentesi attorno all'espressione sono obbligatorie in quanto fanno parte dell'istruzione if e non dell'espressione. Notate anche che, a differenza di quello che accade in altri linguaggi di programmazione, dopo le parentesi non compare la parola then.

Quando un'istruzione if viene eseguita, l'espressione all'interno delle parentesi viene calcolata. Se il valore dell'espressione è diverso da zero (valore che il C interpreta come vero) allora l'istruzione dopo le parentesi viene eseguita. Ecco un esempio:

```
if (line_num == MAX_LINES)
    line_num = 0;
```

L'istruzione `line_num = 0;` viene eseguita se la condizione `line_num == MAX_LINES` è vera (cioè ha valore diverso da zero).



Non confondete l'operatore `==` (uguaglianza) con l'operatore `=` (assegnazione). L'istruzione

```
if (i == 0) ..
```

controlla se i è uguale a 0, mentre l'istruzione

```
if (i = 0) ..
```

assegna 0 a i e poi controlla se il risultato dell'espressione è diverso da zero. In questo caso il test sull'espressione ha sempre esito negativo.

Confondere l'operatore `==` con l'operatore `=` è uno degli errori più comuni nella programmazione C, probabilmente questo è dovuto al fatto che in matematica il simbolo `=` significa "è uguale a" (e lo stesso vale per certi linguaggi di programmazione). Alcuni compilatori generano un messaggio di warning se trovano un `=` dove normalmente dovrebbe esserci un `==`.



Spesso l'espressione contenuta in un'istruzione if ha il compito di controllare se una variabile ricade all'interno di un intervallo di valori. Per esempio per controllare se $0 \leq i < n$ scriveremo

```
if (0 <= i && i < n) ..
```

Per testare la condizione opposta (i è al di fuori di un intervallo di valori), scriveremo:

```
if (i < 0 || i >= n) ..
```

Notate l'uso dell'operatore `||` al posto dell'operatore `&&`.

Le istruzioni composte

Osservate che nel nostro modello dell'istruzione if la parola *istruzione* è singolare e non plurale:

```
if (espressione) istruzione
```

Come potremmo fare se volessimo eseguire due o più istruzioni con un'istruzione if? Questo è il punto dove entra in gioco l'**istruzione composta** (*compound statement*). Un'istruzione composta ha la forma



Racchiudendo tra parentesi graffe un gruppo di istruzioni possiamo farzare il compilatore a trattarle come una istruzione singola.

Ecco un esempio di istruzione composta:

```
{ line_num = 0; page_num++; }
```

Soltamente, per ragioni di chiarezza, scriveremo un'istruzione composta su più righe, mettendo un'istruzione per riga:

```
{
    line_num = 0;
    page_num++;
}
```

Osservate che ogni istruzione interna termina ancora con un punto e virgola, mentre non è così per l'istruzione composta stessa.

Ecco come appare un'istruzione composta quando utilizzata all'interno di un'istruzione if:

```
if (line_num == MAX_LINES){
    line_num = 0;
    page_num++;
}
```

Le istruzioni composte sono comuni anche nei cicli e in tutti i punti in cui la sintassi del C richiede una singola istruzione ma se ne vuole inserire più di una.

La clausola else

L'istruzione if può avere una clausola else:

```
if (espressione) istruzione else istruzione
```

L'istruzione che segue la parola else viene eseguita se l'espressione contenuta tra le parentesi ha valore 0.

Ecco un esempio di un'istruzione if con la clausola else:

```
if (i > j)
    max = i;
else
    max = j;
```

Osservate che entrambe le istruzioni interne terminano con un punto e virgola.

Quando un'istruzione if contiene una clausola else si apre un problema di impiantazione: dove dovrebbe essere messa la clausola else? Molti programmatore C la

allineano con l'if iniziale, così come si vede nell'esempio precedente. Le istruzioni interne di solito vengono indentate, ma se sono corte possono essere posizionate sulla stessa linea delle parole if ed else:

```
if (i > j) max = i;  
else max = j;
```

Non ci sono restrizioni sul tipo di istruzioni che possono apparire all'interno di un costrutto if. Infatti non è insolito che un'istruzione if venga annidata all'interno di un'altra istruzione if. Considerate la seguente istruzione che trova il più grande tra i numeri memorizzati in i, j e k e salva tale valore in max:

```
if (i > j)  
    if (i > k)  
        max = i;  
    else  
        max = k;  
else  
    if (j > k)  
        max = j;  
    else  
        max = k;
```

Le istruzioni if possono essere annidate fino a raggiungere qualsiasi profondità. Osservate come allineare ogni else con il corrispondente if renda gli annidamenti molto più facilmente individuabili. Se nonostante questo gli annidamenti vi sembrano ancora confusi, non esitate ad aggiungere delle parentesi graffe:

```
if (i > j) {  
    if (i > k)  
        max = i;  
    else  
        max = k;  
} else {  
    if (j > k)  
        max = j;  
    else  
        max = k;  
}
```

Aggiungere parentesi graffe alle istruzioni anche quando non sono necessarie è come utilizzare le parentesi nelle espressioni: entrambe le tecniche aiutano a rendere il programma più leggibile e allo stesso tempo scongiurano il rischio che il compilatore interpreti il programma in modo diverso dal nostro.

Alcuni programmatore utilizzano tante parentesi graffe quante se ne possono mettere nelle istruzioni if (e allo stesso modo nei costrutti di iterazione). Un programmatore che adotta questa convenzione includerà un paio di parentesi graffe per ogni clausola if e per ogni clausola else:

```

if (i > j) {
    if (i > k) {
        max = i;
    } else {
        max = k;
    }
} else {
    if (j > k) {
        max = j;
    } else {
        max = k;
    }
}

```

Utilizzare le parentesi graffe anche quando non è necessario presenta due vantaggi. Per prima cosa il programma diventa più facile da modificare perché risulta più agevole l'aggiunta di ulteriori istruzioni alle clausole if ed else.

Istruzioni if in cascata

Spesso abbiamo bisogno di testare una serie di condizioni fermandoci non appena una di queste è vera. Una "cascata" di istruzioni if molte volte è il modo migliore per scrivere questa serie di test. Per esempio le seguenti istruzioni if in cascata controllano se n è minore, uguale o maggiore a 0:

```

if (n < 0)
    printf("n is less than 0\n");
else
    if (n == 0)
        printf("n is equal to 0\n");
    else
        printf("n is greater than 0\n");

```

Sebbene il secondo if sia annidato all'interno del primo, di solito non viene indentato dai programmati C. Questi allineano invece ogni else con il relativo if:

```

if (n < 0)
    printf("n is less than 0\n");
else if (n == 0)
    printf("n is equal to 0\n");
else
    printf("n is greater than 0\n");

```

Questa sistemazione conferisce agli if in cascata una veste distintiva:

```

if ( espressione )
    istruzione
else if ( espressione )
    istruzione
...
else if ( espressione )
    istruzione
else
    istruzione

```

Le ultime due righe (*else istruzione*) non sono sempre presenti ovviamente. Questo stile di indentazione evita il problema delle indentazioni eccessive nei casi in cui il numero di test risulta considerevole. Inoltre, assicura il lettore che il costrutto non è altro che una serie di test.

Tenete in mente che la cascata di if non è un nuovo tipo di istruzione. È semplicemente un'ordinaria istruzione if che ha un'altra istruzione if come sua clausola else (e quell'istruzione if ha a sua volta un'altra istruzione if come sua clausola else e così via all'infinito).

PROGRAMMA

Calcolare le commissioni dei broker

Quando delle azioni vengono vendute o comperate attraverso un broker finanziario, la commissione del broker viene calcolata utilizzando una scala mobile che dipende dal valore delle azioni scambiate. Diciamo che le commissioni di un broker corrispondano a quelle illustrate nella seguente tabella:

Dimensione della transazione	Commissione
Sotto i 2.500\$	30\$ + 1,7%
2.500\$ – 6.250\$	56\$ + 0,66%
6.250\$ – 20.000\$	76\$ + 0,34%
20.000\$ – 50.000\$	100\$ + 0,22%
50.000\$ – 500.000\$	155\$ + 0,11%
Oltre i 500.000\$	255\$ + 0,09%

La tariffa minima è di 39\$. Il nostro prossimo programma chiederà all'utente di immettere l'ammontare delle azioni scambiate per poi visualizzare il valore della relativa commissione:

Enter value of trade: 30000
Commission: \$166.00

Il cuore del programma è una cascata di istruzioni if che determina in quale intervallo ricade lo scambio di azioni.

```

broker.c /* Calcola la commissione di un broker */

#include <stdio.h>

int main(void)
{
    float commission, value;

    printf("Enter value of trade: ");
    scanf("%f", &value);

    if (value < 2500.00f)
        commission = 30.00f + .017f * value;
    else if (value < 6250.00f)
        commission = 56.00f + .0066f * value;
    else if (value < 20000.00f)
        commission = 76.00f + .0034f * value;
    else if (value < 50000.00f)
        commission = 100.00f + .0022f * value;
    else if (value < 500000.00f)
        commission = 155.00f + .0011f * value;
    else
        commission = 255.00f + .0009f * value;

    if (commission < 39.00f)
        commission = 39.00f;

    printf("Commission: $%.2f\n", commission);

    return 0;
}

```

Gli if in cascata avrebbero potuto essere scritti in questo modo (le modifiche sono scritte in **grassetto**):

```

if (value < 2500.00f)
    commission = 30.00f + .017f * value;
else if (value >= 2500.00f && value < 6250.00f)
    commission = 56.00f + .0066f * value;
else if (value >= 6250.00f && value < 20000.00f)
    commission = 76.00f + .0034f * value;

```

Nonostante il programma continua a funzionare, le condizioni inserite non sono necessarie. Per esempio: la prima clausola if controlla se il valore è minore di 2500 e in quel caso calcola la commissione.

Quando raggiungiamo l'espressione del secondo if (`value >= 2500.00f && value < 6250.00f`) sappiamo già che value non può essere inferiore a 2500 e quindi deve essere maggiore o uguale a 2500. La condizione `value >= 2500.00f` sarà sempre vera e quindi non c'è nessun bisogno di controllarla.

Il problema dell'else pendente

Quando istruzioni if vengono annidate dobbiamo fare attenzione al noto problema dell'else pendente (*dangling else*). Considerate l'esempio seguente:

```
if (y != 0)
    if (x != 0)
        result = x / y;
else
    printf("Error: y is equal to 0\n");
```

A quale if appartiene la clausola else? L'indentazione suggerisce che appartiene all'istruzione if più esterna. Tuttavia il C segue la regola che impone che una clausola else appartenga all'istruzione if più vicina che non sia già accoppiata con un else. In questo esempio la clausola else appartiene all'istruzione if più interna. Quindi una versione correttamente indentata sarebbe:

```
if (y != 0)
    if (x != 0)
        result = x / y;
else
    printf("Error: y is equal to 0\n");
```

Per fare in modo che la clausola else faccia parte dell'istruzione if più esterna possiamo racchiudere tra parentesi graffe l'if più interno:

```
if (y != 0) {
    if (x != 0)
        result = x / y;
} else
    printf("Error: y is equal to 0\n");
```

Questo esempio prova il valore delle parentesi all'interno dei programmi. Se le avessimo usate sin dal principio con l'if originale, non avremmo incontrato problemi.

Espressioni condizionali

L'istruzione if del C permette a un programma di eseguire una o più azioni a seconda del valore di un'espressione. Inoltre il C prevede un operatore che permette a un'espressione di produrre uno tra due valori a seconda del valore assunto da un'altra espressione.

L'operatore condizionale consiste di due simboli (?) e (:) che devono essere utilizzati congiuntamente in questo modo:

```
expr1 ? expr2 : expr3
```

expr1, expr2 ed expr3 possono essere espressioni di qualsiasi tipo. L'espressione risultante viene detta **espressione condizionale**. L'operatore condizionale è unico tra gli operatori C in quanto richiede tre operandi invece che uno o due. Per questa ragione spesso gli si fa riferimento come all'operatore **ternario**.

L'espressione condizionale $expr1 ? expr2 : expr3$ dovrebbe essere letta come "se $expr1$ allora $expr2$ altrimenti $expr3$ ". L'espressione viene valutata in vari stadi: $expr1$ viene calcolata per prima, se il suo valore è diverso da zero allora viene calcolata $expr2$, il cui valore sarà quello dell'intera espressione condizionale. Se il valore di $expr1$ è zero allora l'espressione condizionale assumerà il valore di $expr3$.

L'esempio seguente illustra l'operatore condizionale:

```
int i, j, k;
i = 1;
j = 2;
k = i > j ? i : j;           /* adesso k è uguale a 2 */
k = (i >= 0 ? i : 0) + j   /* adesso k è uguale a 3 */
```

L'espressione condizionale $i > j ? i : j$ nella prima assegnazione a k ritorna il valore di i o quello di j a seconda di quale tra questi sia il maggiore. Dato che i vale 1 e j vale 2, il confronto $i > j$ da esito negativo e il valore dell'espressione condizionale che viene assegnato a k è 2. Nella seconda assegnazione a k il confronto $i \geq 0$ ha esito positivo e quindi l'espressione $(i \geq 0 ? i : 0)$ ha valore 1, il quale viene sommato a j producendo il valore 3. Le parentesi sono necessarie, infatti l'ordine di precedenza dell'operatore condizionale è minore di quello degli altri operatori che abbiamo discusso fino a ora, fatta eccezione per l'operatore di assegnazione.

Le espressioni condizionali tendono a rendere i programmi più corti ma anche più difficili da comprendere, molto probabilmente è meglio evitarle. Nonostante questo ci sono un paio di occasioni in cui il loro utilizzo può essere accattivante: uno di queste è l'istruzione return. Invece di scrivere

```
if (i > j)
    return i;
else
    return j;
```

molti programmatori scriverebbero

```
return i > j ? i : j;
```

Anche le chiamate alla printf possono beneficiare in certi casi delle espressioni condizionali. Al posto di

```
if (i > j)
    printf("%d\n", i);
else
    printf("%d\n", j);
```

possiamo scrivere semplicemente

```
printf("%d\n", i > j ? i : j);
```

Le espressioni condizionali sono spesso comuni anche in certi tipi di definizioni di macro [definizioni di macro > 14.3].

Valori booleani nel C89

Per molti anni il linguaggio C ha sofferto della mancanza di uno specifico tipo booleano, che per altro non è definito nemmeno nello standard C89. Questa omissione finisce per essere una limitazione visto che molti programmati hanno bisogno di variabili che siano in grado di memorizzare valori come *falso* e *vero*. Un modo per aggirare questa limitazione del C89 è quella di dichiarare una variabile int e di assegnarle i valori 0 o 1:

```
int flag;  
flag = 0;  
-  
flag = 1;
```

Sebbene questo schema funzioni non contribuisce molto alla leggibilità del programma. Non è ovvio che a flag debbano essere assegnati solamente i valori booleani, né che 0 e 1 rappresentino rispettivamente falso e vero.

Per rendere i programmi più comprensibili, i programmati C89 definiscono spesso delle macro con nomi come TRUE e FALSE:

```
#define TRUE 1  
#define FALSE 0
```

Adesso l'assegnazione a flag ha un'apparenza molto più naturale:

```
int flag;  
flag = FALSE;  
-  
flag = TRUE;
```

Per testare se la variabile flag contiene un valore corrispondente a true, possiamo scrivere:

```
if (flag == TRUE) ...  
oppure più semplicemente  
if (flag) ...
```

l'ultima scrittura è migliore non solo perché più concisa, ma anche perché continuerebbe a funzionare correttamente anche se flag avesse valori diversi da 0 e 1.

Per testare se la variabile flag contiene un valore corrispondente a false, possiamo scrivere:

```
if (flag == FALSE) ...  
oppure  
if (!flag) ...
```

Proseguendo con quest'idea possiamo anche pensare di definire una macro che possa essere utilizzata come tipo:

```
#define BOOL int
```

`BOOL` può prendere il posto di `int` quando devono essere dichiarate delle variabili booleane:

```
BOOL flag;
```

Adesso è chiaro che la variabile `flag` non è una variabile `int` ordinaria ma rappresenta una condizione booleana (naturalmente il compilatore continuerà a trattare `flag` come una variabile `int`). Nei prossimi capitoli scopriremo che con il C89 ci sono metodi migliori per dichiarare un tipo booleano utilizzando la definizione di tipo e le enumerazioni [definizione di tipi > 7.5] [enumerazioni > 16.5].

C99

Valori booleani in C99

D&R

La lunga mancanza di un tipo booleano è stata rimediata nel C99, il quale prevede il tipo `_Bool`. In questa versione del C una variabile booleana può essere dichiarata scrivendo

```
_Bool flag;
```

`_Bool` è un tipo di intero (più precisamente un tipo di intero senza segno [tipi di interi senza segno > 7.1]) e quindi non è altro che una variabile intera camuffata. Tuttavia, a differenza delle variabili intere ordinarie, a una variabile `_Bool` possono essere assegnati solo i valori 0 o 1. Più genericamente, cercare di memorizzare un valore diverso da zero in una variabile `_Bool` fa sì che alla variabile venga assegnato il valore 1:

```
flag = 5; /* a flag viene assegnato il valore 1 */
```

È ammesso (anche se non consigliabile) eseguire dei calcoli aritmetici con le variabili `_Bool`. È anche possibile stampare una variabile `_Bool` (verrà visualizzato 0 o 1). Naturalmente il valore di una variabile `_Bool` può essere controllato all'interno di un'istruzione `if`:

```
if (flag) /* controlla se flag è uguale a 1 */
```

Oltre alla definizione del tipo `_Bool`, il C99 fornisce il nuovo header `<stdbool.h>` [header <stdbool.h> > 21.5], il quale agevola l'utilizzo dei valori booleani. Questo header fornisce la macro `bool` che corrisponde a `_Bool`. Se `<stdbool.h>` è stato incluso allora possiamo scrivere

```
bool flag; /* come scrivere _Bool flag */
```

L'header `<stdbool.h>` fornisce anche delle macro chiamate `true` e `false` che corrispondono rispettivamente a 1 e 0, rendendo possibile scrivere:

```
flag = false;
```

```
flag = true;
```

Dato che l'header `<stdbool.h>` è così comodo, lo utilizzeremo nei programmi seguenti ogni volta che delle variabili booleane si riveleranno necessarie.

5.3 L'istruzione switch

Nella programmazione di tutti i giorni abbiamo spesso bisogno di confrontare un'espressione con una serie di valori per vedere a quale di questi corrisponda. Nella Sezione 5.2 abbiamo visto che a questo scopo possono essere utilizzate delle istruzioni if in cascata. Per esempio i seguenti if in cascata stampano le parole inglese corrispondenti ai voti numerici:

```
if (grade == 4)
    printf("Excellent");
else if (grade == 3)
    printf("Good");
else if (grade == 2)
    printf("Average");
else if (grade == 1)
    printf("Poor");
else if (grade == 0)
    printf("Failing");
else
    printf("Illegal grade");
```

Come alternativa a questa cascata di istruzioni if, il C prevede l'istruzione switch. Il costrutto switch seguente è equivalente alla nostra cascata di if:

```
switch (grade) {
    case 4: printf("Excellent");
              break;
    case 3: printf("Good");
              break;
    case 2: printf("Average");
              break;
    case 1: printf("Poor");
              break;
    case 0: printf("Failing");
              break;
    default: printf("Illegal grade");
              break;
}
```

Quando questa istruzione viene eseguita il valore della variabile grade viene confrontato con 4, 3, 2, 1 e 0. Se per esempio corrisponde a 4 allora viene stampato il messaggio Excellent e successivamente l'istruzione break si occupa di trasferire il controllo all'istruzione che segue lo switch. Se il valore di grade non corrisponde a nessuno dei codici elencati allora viene applicato il caso default e quindi viene stampato il messaggio Illegal grade.

Un'istruzione switch è spesso più facile da leggere rispetto a degli if in cascata. Inoltre le istruzioni switch spesso risultano più veloci in esecuzione, specialmente se ci sono parecchi casi.

Nella sua forma più comune l'istruzione switch si presenta in questo modo:

```
switch (espressione) {
    case espressione-costante: istruzioni
    case espressione-costante: istruzioni
    default: istruzioni
}
```

L'istruzione switch è piuttosto complessa. Diamo un'occhiata alle sue componenti una alla volta:

- **Espressioni di controllo.** La parola switch deve essere seguita da un'espressione intera racchiusa tra parentesi. Nel C i caratteri vengono trattati come numeri interi e quindi possono essere confrontati nelle istruzioni switch. I numeri a virgola mobile e le stringhe invece non sono utilizzabili.
- **Etichette case.** Ogni caso inizia con un'etichetta della forma
`case espressione-costante :`
 Un'espressione costante è praticamente come una normale espressione a eccezione del fatto che non può contenere variabili o chiamate a funzione. Quindi, 5 è un'espressione costante, così come lo è $5 + 10$, mentre non lo è $n + 10$ (a meno che n non sia una macro che rappresenta una costante). L'espressione costante in un'etichetta case deve restituire un intero (sono accettabili anche i caratteri).
- **Istruzioni.** Dopo ogni etichetta case può esserci un numero qualsiasi di istruzioni. Attorno a queste istruzioni non è necessaria nessuna parentesi graffa. Normalmente break è l'ultima istruzione di ogni gruppo.

Non sono ammesse etichette case duplicate, non ha importanza invece l'ordine con cui sono disposti i casi. In particolare il caso default non deve essere necessariamente l'ultimo.

Al seguito della parola case può esserci una sola espressione costante, tuttavia diverse etichette case possono precedere lo stesso gruppo di istruzioni:

```
switch (grade) {
    case 4:
    case 3:
    case 2:
    case 1: printf("Passing");
    break;
    case 0: printf("Failing");
    break;
    default: printf("Illegal grade");
    break;
}
```

A volte, al fine di risparmiare spazio, i programmati mettono diverse etichette case sulla stessa riga:

```

switch (grade) {
    case 4: case 3: case 2: case 1:
        printf("Passing");
        break;
    case 0: printf("Failing");
        break;
    default: printf("Illegal grade");
        break;
}

```

Sfortunatamente non c'è modo di scrivere un'etichetta che specifichi un intervallo di valori come avviene in alcuni linguaggi di programmazione.

Un'istruzione switch non necessita di un caso default. Se default manca e il valore dell'espressione di controllo non combacia con nessuno dei casi, allora il controllo passa semplicemente all'istruzione che segue lo switch.

Il ruolo dell'istruzione break

Vediamo ora con maggiore attenzione l'istruzione break. Come abbiamo visto in precedenza, l'esecuzione dell'istruzione break causa l'uscita del programma dal costrutto switch per passare all'istruzione successiva.

La ragione per la quale l'istruzione break è necessaria, è dovuta al fatto che l'istruzione switch è in realtà una forma di *salto precalcolato*. Quando l'espressione di controllo viene calcolata, il programma salta all'etichetta corrispondente al valore dell'espressione. Queste etichette non sono altro che un segno indicante una posizione all'interno del costrutto switch. Quando l'ultima istruzione del caso è stata eseguita, il controllo passa alla prima istruzione del caso successivo ignorando completamente l'etichetta case. Senza break (o qualche altra istruzione di salto), il controllo passerebbe da un caso a quello successivo. Considerate il seguente costrutto switch:

```

switch (grade) {
    case 4: printf("Excellent");
    case 3: printf("Good");
    case 2: printf("Average");
    case 1: printf("Poor");
    case 0: printf("Failing");
    default: printf("Illegal grade");
}

```

Se grade ha valore 3, il messaggio che viene stampato è

GoodAveragePoorFailingIllegal grade



Dimenticare l'istruzione break è un errore comune. Sebbene l'omissione di break in certe situazioni sia intenzionale al fine di permettere la condivisione del codice tra più casi, solitamente non è altro che una svista.

Visto che passare da un caso al successivo raramente viene fatto in modo deliberato, è una buona norma segnalare esplicitamente questi casi di omissione dell'istruzione break:

```
switch (grade) {
    case 4: case 3: case 2: case 1:
        num_passing++;
        /* CONTINUA CON IL CASE SUCCESSIVO */
    case 0: total_grades++;
        break;
}
```

Senza l'aggiunta del commento qualcuno potrebbe successivamente correggere l'"errore" aggiungendo un'istruzione break non voluta.

Sebbene l'ultimo caso dell'istruzione switch non necessiti mai dell'istruzione break, è pratica comune inserirlo comunque. Questo viene fatto come difesa dal problema del "break mancate" qualora in un secondo momento si dovessero inserire degli altri casi.

PROGRAMMA

Stampare la data nel formato legale

I contratti e altri documenti legali vengono spesso datati nel seguente modo:

Dated this _____ day of _____, 20____.

Scriviamo un programma che visualizza le date in questa forma. Faremo immettere la data all'utente nel formato anglosassone mese/giorno/anno e successivamente visualizzeremo la stessa data nel formato "legale":

Enter date (mm/dd/yy) : 7/19/14
Dated this 19th day of July, 2014.

Possiamo utilizzare la printf per la maggior parte della formattazione. Tuttavia rimangono due problemi: come aggiungere al giorno il suffisso "th" (o "st" o "nd" o "rd"), e come indicare il mese con una parola invece che con un numero. Per fortuna l'istruzione switch è l'ideale per entrambe le situazioni: useremo uno switch per il suffisso del giorno e un altro per il nome del mese.

```
date.c /* Stampa la data nel formato legale */
#include <stdio.h>

int main(void)
{
    int month, day, year;
    printf("Enter date (mm/dd/yy): ");
    scanf("%d /%d /%d", &month, &day, &year);
    printf("Dated this %d", day);
    switch (day) {
```

```

        case 1: case 21: case 31:
            printf("st"); break;
        case 2: case 22:
            printf("nd"); break;
        case 3: case 23:
            printf("rd"); break;
        default: printf("th"); break;
    }
    printf(" day of ");
    switch (month) {
        case 1: printf("January"); break;
        case 2: printf("February"); break;
        case 3: printf("March"); break;
        case 4: printf("April"); break;
        case 5: printf("May"); break;
        case 6: printf("June"); break;
        case 7: printf("July"); break;
        case 8: printf("August"); break;
        case 9: printf("September"); break;
        case 10: printf("October"); break;
        case 11: printf("November"); break;
        case 12: printf("December"); break;
    }
    printf(", %02d.\n", year);
    return 0;
}

```

Fate caso all'uso di `%2d` per la visualizzazione delle ultime due cifre dell'anno. Se avessimo utilizzato `%d` al suo posto, allora gli anni con singola cifra verrebbero visualizzati in modo sbagliato (2005 verrebbe visualizzato come 205).

Domande e risposte

D: Molti compilatori non producono messaggi di warning quando viene utilizzato `=` al posto di `==`. C'è qualche modo per forzare il compilatore a notare il problema? [p. 79]

R: Alcuni programmatore utilizzano un trucco: per abitudine invece di scrivere

```
if (i == 0) -
```

scrivono

```
if (0 == i) -
```

Supponete adesso che al posto dell'operatore `==` venga accidentalmente scritto `=`:

```
if (0 = i) -
```

In tal caso il compilatore produrrà un messaggio di errore visto che non è possibile assegnare un valore a 0. Noi non utilizzeremo questo trucchetto perché rende l'aspetto dei programmi un po' innaturale. Inoltre, può essere usato solo nel caso in cui nella condizione di controllo uno dei due operandi non è un lvalue.

Fortunatamente molti compilatori sono in grado di controllare l'uso sospetto dell'operatore = all'interno delle condizioni degli if. Il compilatore GCC per esempio, effettua questo controllo se viene utilizzata l'opzione -Wparentheses, oppure se viene selezionata l'opzione -Wall (tutti i warning). Inoltre, GCC permette ai programmati-ri di sovrapporre i messaggi di warning nei casi in cui fosse necessario, richiudendo la condizione if all'interno di un secondo set di parentesi:

```
if ((i = j)) -
```

D: I libri sul C sembrano adottare diversi stili di indentazione e posizionamento delle parentesi graffe per l'istruzione composta. Qual è lo stile migliore?

R: Secondo il libro *The New Hacker's Dictionary* (Cambridge, Mass.: MIT Press, 1996), comunemente vengono utilizzati quattro stili di indentazione e di disposizione delle parentesi:

- Lo stile K&R utilizzato nel libro *The C Programming Language* di Kernighan e Ritchie. È lo stile utilizzato nei programmi di questo libro. Nello stile K&R la parentesi graffa sinistra appare alla fine di una riga:

```
if (line_num == MAX_LINES) {
    line_num = 0;
    page_num++;
}
```

Non mettendo la parentesi sinistra su una riga a se stante, lo stile K&R mantiene i programmi più compatti. Uno svantaggio è che la parentesi graffa sinistra può diventare difficile da trovare (personalmente non vedo questo come un problema in quanto l'indentazione delle istruzioni interne rende chiaro dove dovrebbe trovarsi la parentesi). Tra l'altro, lo stile K&R è uno dei più utilizzati in Java.

- Lo stile Allman, il cui nome deriva da Eric Allman (l'autore di sendmail e altre utility UNIX), mette la parentesi graffa sinistra su una riga a se stante:

```
if (line_num == MAX_LINES)
{
    line_num = 0;
    page_num++;
}
```

Questo stile rende più facile controllare che le parentesi immesse siano sempre a coppie.

- Lo stile Whitesmiths, reso popolare dal compilatore C Whitesmiths, impone che le parentesi graffe debbano essere indentate:

```

if (line_num == MAX_LINES)
{
    line_num = 0;
    page_num++;
}

```

- Lo stile GNU, utilizzato nel software prodotto per lo GNU Project, indenta le parentesi e indenta ulteriormente le istruzioni interne:

```

if (line_num == MAX_LINES)
{
    line_num = 0;
    page_num++;
}

```

Quale stile utilizzare è solo questione di gusti: non ci sono prove che uno stile sia migliore rispetto agli altri. In ogni caso, scegliere lo stile corretto è meno importante che applicare quest'ultimo costantemente.

D: Se i è una variabile int ed f è una variabile float, di che tipo sarà l'espressione condizionale (i > 0 ? i : f) ?

R: Quando, come avviene nell'esempio, valori int e float vengono mischiati all'interno di una espressione condizionale, quest'ultima sarà di tipo float. Se l'espressione i > 0 è vera, allora il suo valore sarà pari al valore di i convertito al tipo float.

D: Perché il C99 non ha un nome migliore per il suo tipo booleano? [p. 88]

R: _Bool non è un nome molto elegante. Nomi molto più comuni come bool o boolean non sono stati scelti in quanto i programmi C già esistenti avrebbero potuto aver già definito questi nomi e questo avrebbe comportato la mancata compilazione del vecchio codice.

D: OK, ma allora perché il nome _Bool non dovrebbe interferire allo stesso modo con i vecchi programmi?

R: Lo standard C89 specifica che i nomi che cominciano con un underscore seguito da una lettera maiuscola sono riservati per scopi futuri e quindi non devono essere utilizzati dai programmati.

***D: Il modello illustrato per l'istruzione switch è stato descritto come quello per la "forma più comune". Ci sono altre forme di utilizzo dell'istruzione? [p.89]**

R: L'istruzione switch ha una forma più generale di quella descritta in questo capitolo, tuttavia la descrizione fornita qui è virtualmente sufficiente per tutti i programmi. [etichette > 6.4] Per esempio, un'istruzione switch può contenere etichette che non sono precedute dalla parola case, il che conduce a una trappola. Supponete di scrivere accidentalmente la parola default in modo non corretto:

```

switch (...) {
    default: ...
}

```

Il compilatore potrebbe non rilevare l'errore in quanto potrebbe assumere che default sia una semplice etichetta.

D: Ho visto diversi metodi per indentare l'istruzione switch. Qual è il migliore?

R: Ci sono almeno due metodi. Il primo è quello di mettere le istruzioni di ogni caso *dopo* l'etichetta:

```
switch (coin) {
    case 1: printf("Cent");
    break;
    case 5: printf("Nikel");
    break;
    case 10: printf("Dime");
    break;
    case 25: printf("Quarter");
    break;
}
```

Se ogni caso consiste di una singola azione (in questo esempio una chiamata alla printf), allora l'istruzione break può anche andare sulla stessa linea di azione:

```
switch (coin) {
    case 1: printf("Cent"); break;
    case 5: printf("Nikel"); break;
    case 10: printf("Dime"); break;
    case 25: printf("Quarter"); break;
}
```

L'altro metodo è quello di mettere le istruzioni sotto l'etichetta indentandole per far risaltare quest'ultima:

```
switch (coin) {
    case 1:
        printf("Cent");
        break;
    case 5:
        printf("Nikel");
        break;
    case 10:
        printf("Dime");
        break;
    case 25:
        printf("Quarter");
        break;
}
```

Una variante di questo schema prevede che ogni etichetta sia allineata sotto la parola switch.

Il primo metodo è indicato per le situazioni in cui le istruzioni contenute in ogni caso sono poche e brevi. Il secondo metodo è più indicato per grandi strutture switch dove le istruzioni presenti nei vari casi siano numerose e/o complesse.

Esercizi

Sezione 5.1

1. I seguenti frammenti di programma illustrano gli operatori relazionali e di uguaglianza. Mostrate l'output prodotto da ognuno assumendo che i, j e k siano variabili int.

(a) `i = 2; j = 3;
k = i * j == 6;
printf("%d", k);`

(b) `i = 5; j = 10; k = 1;
printf("%d", k > 1 < j);`

(c) `i = 3; j = 2; k = 1;
printf("%d", i < j == j < k);`

(d) `i = 3; j = 4; k = 5;
printf("%d", i % j + i < k);`

2. I seguenti frammenti di programma illustrano gli operatori logici. Mostrate l'output prodotto da ognuno assumendo che i, j e k siano variabili int.

(a) `i = 10; j = 5;
printf("%d", !i < j);`

(b) `i = 2; j = 1;
printf("%d", !!i + !j);`

(c) `i = 5; j = 0; k = -5;
printf("%d", i && j || k);`

(d) `i = 1; j = 2; k = 3;
printf("%d", i < j || k);`

3. *I seguenti frammenti di programma illustrano il comportamento di corto circuitazione delle espressioni logiche. Mostrate l'output prodotto da ognuno assumendo che i, j e k siano variabili int.

(a) `i = 3; j = 5; k = 5;
printf("%d ", i < j || ++j < k);
printf("%d %d %d", i, j, k);`

(b) `i = 7; j = 8; k = 9;
printf("%d ", i - 7 && j++ < k);
printf("%d %d %d", i, j, k);`

(c) `i = 7; j = 8; k = 9;
printf("%d ", (i = j) || (j = k));
printf("%d %d %d", i, j, k);`

(d) `i = 1; j = 1; k = 1;
printf("%d ", ++i || ++j && ++k);
printf("%d %d %d", i, j, k);`

- W 4. *Scrivete una singola espressione il cui valore possa essere sia -1 che 0 o +1 a seconda che il valore di i sia rispettivamente minore, uguale o maggiore di quello di j.

Sezione 5.2 5. *La seguente istruzione if è ammissibile?

```
if (n >= 1 <= 10)
    printf("n is between 1 and 10\n");
```

Nel caso lo fosse, cosa succede se il valore di n è uguale a 0?

6. *La seguente istruzione if è ammissibile?

```
if (n == 1-10)
    printf("n is between 1 and 10\n");
```

Nel caso lo fosse, cosa succede se il valore di n è uguale a 5?

7. Che cosa stampa la seguente istruzione se i ha il valore 17? E cosa viene visualizzato invece se i ha il valore -17?

```
printf("%d\n", i >= 0 ? i : -i);
```

8. La seguente istruzione if è inutilmente complicata. Semplificatela il più possibile (Suggerimento: l'intera istruzione può essere rimpiazzata da una singola assegnazione).

```
if (age >= 13)
    if (age <= 19)
        teenager = true;
    else
        teenager = false;
else if (age < 13)
    teenager = false;
```

9. Le seguenti istruzioni if sono equivalenti? Se no, perché?

<pre>if (score >= 90) printf("A"); else if (score >= 80) printf("B"); else if (score >= 70) printf("C"); else if (score >= 60) printf("D"); else printf("F");</pre>	<pre>if (score < 60) printf("F"); else if (score < 70) printf("D"); else if (score < 80) printf("C"); else if (score < 90) printf("B"); else printf("A");</pre>
---	---

Sezione 5.3 10. *Che output produce il seguente frammento di programma? (Assumete che i sia una variabile intera).

```
i = 1;
switch (i % 3) {
    case 0: printf("zero");
    case 1: printf("one");
    case 2: printf("two");
}
```

11. La tabella seguente mostra i codici telefonici delle aree appartenenti allo stato della Georgia unitamente alla città di più grandi dimensioni presente nell'area stessa:

Prefisso	Città Principale
229	Albany
404	Atlanta
470	Atlanta
478	Macon
678	Atlanta
706	Columbus
762	Columbus
770	Atlanta
912	Savannah

Scrivete un costrutto switch che abbia come espressione di controllo la variabile area_code. Se il valore di area_code è presente nella tabella allora l'istruzione switch deve stampare il nome della città corrispondente. In caso contrario l'istruzione switch dovrà visualizzare il messaggio "Area code not recognized". Utilizzate le tecniche discusse nella Sezione 5.3 per rendere l'istruzione switch la più semplice possibile.

Progetti di programmazione

1. Scrivete un programma che calcoli quante cifre sono contenute in un numero:

```
Enter a number: 374
The number 374 has 3 digits
```

Potete assumere che il numero non abbia più di quattro cifre. Suggerimento: usate l'istruzione if per controllare il numero. Per esempio, se il numero è tra 0 e 9 allora ha una sola cifra. Se il numero è tra 10 e 99 allora ha due cifre.

2. Scrivete un programma che chieda all'utente un orario nel formato a 24 ore e successivamente visualizzi lo stesso orario nel formato a 12 ore:

```
Enter a 24-hour time: 21:11
Equivalent 12-hour time: 9:11 PM
```

Fate attenzione a non visualizzare 12:00 come 0:00.

3. Modificate il programma broker.c della Sezione 5.2 applicando le seguenti modifiche:

- Chiedere all'utente di immettere un numero di azioni e il prezzo per azione invece di chiedere il valore dello scambio.
- Aggiungere le istruzioni per il calcolo della commissione di un broker rivale (33\$ e 3¢ ad azione per un volume inferiore alle 2000 azioni, 33\$ e 2¢ ad azione per un volume pari o superiore alle 200 azioni). Visualizzare sia il valore della commissione del rivale che quella applicata dal broker originale.

- W 4. Ecco una versione semplificata della scala di Beaufort che viene utilizzata per determinare la forza del vento:

Velocità (nodi)	Descrizione
Minore di 1	Calamo
1 - 3	Bava di vento
4 - 27	Brezza
28 - 47	Burrasca
48 - 63	Tempesta
Oltre 63	Uragano

Scrivete un programma che chieda all'utente di immettere un valore di velocità del vento (in nodi) e visualizzi di conseguenza la descrizione corrispondente.

5. In uno Stato i residenti sono soggetti alle seguenti imposte sul reddito:

Reddito	Ammontare imposta
Non superiore a 750\$	1% del reddito
750\$ - 2.250\$	7,50\$ più il 2% della quota sopra i 750\$
2.250\$ - 3.750\$	37,50\$ più il 3% della quota sopra i 2.250\$
3.750\$ - 5.250\$	82,50\$ più il 4% della quota sopra i 3.750\$
5.250\$ - 7.000\$	142,50\$ più il 5% della quota sopra i 5.250\$
Oltre i 7.000\$	230,00\$ più il 6% della quota sopra i 7.000\$

Scrivete un programma che chieda all'utente di immettere il suo reddito imponibile e successivamente visualizzi l'imposta dovuta.

- W 6. Modificate il programma `upc.c` della Sezione 4.1 in modo da controllare se un codice UPC è valido. Dopo l'immissione del codice UPC da parte dell'utente, il programma dovrà scrivere `VALID` o `NOT VALID`.
7. Scrivete un programma in grado di trovare il minimo e il massimo tra quattro numeri immessi dall'utente:

Enter four integers: 21 43 10 35

Largest: 43

Smallest: 10

Utilizzate il minor numero di istruzioni possibili. Suggerimento: Quattro istruzioni `if` sono sufficienti.

8. La seguente tabella mostra i voli giornalieri tra due città:

Orario Partenza	Orario Arrivo
8:00 a.m.	10:16 a.m.
9:43 a.m.	11:52 a.m.
11:19 a.m.	1:31 a.m.
12:47 p.m.	3:00 p.m.
2:00 p.m.	4:08 p.m.
3:45 p.m.	5:55 p.m.
7:00 p.m.	9:20 p.m.
9:45 p.m.	11:58 p.m.

Scrivete un programma che chieda all'utente di immettere un orario (espresso in ore e minuti utilizzando il formato a 24 ore). Il programma deve visualizzare gli orari di partenza e di arrivo del volo il cui orario di partenza è il più prossimo a quello immesso dall'utente:

Enter a 24-hour time: 13:15

Closest departure time is 12:47 p.m., arriving at 3:00 p.m.

Suggerimento: Convertite l'input in un orario espresso in minuti dalla mezzanotte e confrontatelo con gli orari di partenza, anch'essi espressi come minuti dalla mezzanotte. Per esempio: 13:15 corrisponde a $13 \times 60 + 15 = 795$ minuti dopo la mezzanotte, che è più vicino a 12:37 p.m. (767 minuti dopo la mezzanotte) rispetto a qualsiasi altro orario di partenza.

9. Scrivete un programma che chieda all'utente di immettere due date e che indichi quale delle due si trova prima nel calendario:

Enter first date (mm/dd/yy): 3/6/08

Enter second date (mm/dd/yy): 5/17/07

5/17/07 is earlier than 3/6/08

- W 10. Utilizzate l'istruzione switch per scrivere un programma che converta un voto numerico in un voto espresso attraverso una lettera:

- Enter numerical grade: 84

Letter grade: B

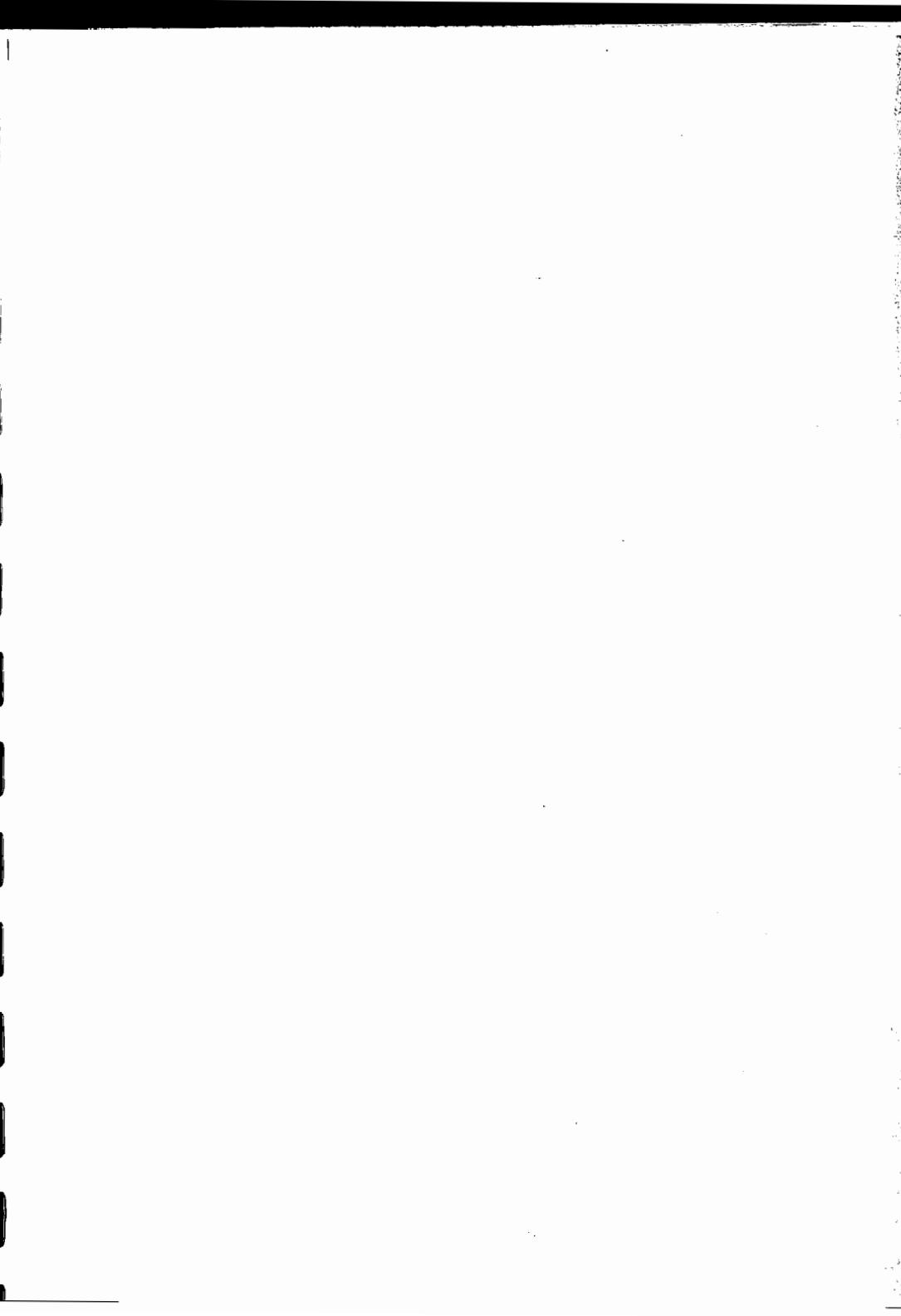
Utilizzate la seguente scala: A = 90-100, B = 80-89, C = 70-79, D = 60-69, F = 0-59. Stampate un messaggio di errore nel caso un cui il voto fosse maggiore di 100 o minore di 0. *Suggerimento:* suddividete il voto in due cifre e poi utilizzate l'istruzione switch per testare la cifra delle decine.

11. Scrivete un programma che chieda all'utente un numero a due cifre e successivamente scriva la dicitura inglese per quel numero:

Enter a two-digit number: 45

You entered the number forty-five.

Suggerimento: suddividete il numero in due cifre. Usate uno switch per stampare la parola corrispondente alla prima cifra ("twenty", "thirty" e così via). Usate un secondo costrutto switch per stampare la parola associata alla seconda cifra. Non dimenticate che i numeri tra 11 e 19 richiedono un trattamento speciale.



6 Cicli

Il Capitolo 5 si è occupato delle istruzioni di selezione if e switch; questo capitolo introduce le istruzioni C per le iterazioni che ci permettono di creare i cicli.

Un **ciclo** è un'istruzione il cui scopo è l'esecuzione ripetitiva di altre istruzioni (**il corpo del ciclo**). In C ogni ciclo possiede un'**espressione di controllo**. Ogni volta che il corpo del ciclo viene eseguito (un'**iterazione** del ciclo), l'espressione di controllo viene analizzata. Se l'espressione è vera (ha valore diverso da zero) allora il ciclo continua nella sua esecuzione.

Il C fornisce tre istruzioni di iterazione: while, do e for, che vengono trattate rispettivamente nelle Sezioni 6.1, 6.2 e 6.3. L'istruzione while viene utilizzata per i cicli la cui espressione di controllo viene analizzata *prima* dell'esecuzione del corpo del ciclo. L'istruzione do invece viene utilizzata per i cicli dove l'espressione di controllo viene analizzata *dopo* l'esecuzione del corpo del ciclo. L'istruzione for è adatta ai cicli che incrementano o decrementano una variabile contatore. La Sezione 6.3 introduce anche l'operatore virgola che viene utilizzato principalmente all'interno delle istruzioni for.

Le ultime due sezioni di questo capitolo sono dedicate alle funzionalità del C utilizzate in abbinamento ai cicli. La Sezione 6.4 descrive le istruzioni break, continue e goto. L'istruzione break fuoriesce da un ciclo e trasferisce il controllo all'istruzione successiva al ciclo stesso. L'istruzione continue salta l'esecuzione della parte rimanente dell'iterazione. L'istruzione goto effettua un salto verso una qualsiasi istruzione presente all'interno di una funzione. La Sezione 6.5 tratta l'istruzione vuota, la quale può essere utilizzata per creare cicli il cui corpo è vuoto.

6.1 L'istruzione while

Di tutti i modi per creare cicli che il linguaggio C ha a disposizione, l'istruzione while è il più semplice e fondamentale. L'istruzione while ha la seguente forma:

```
while (espressione) istruzione
```

All'interno delle parentesi vi è l'espressione di controllo, mentre l'istruzione dopo le parentesi è il corpo del ciclo. A pagina seguente un esempio.

```
while (i < n) /* espressione di controllo */
    i = i * 2; /* corpo del ciclo */
```

Tenete presente che le parentesi sono obbligatorie e che non deve esserci nulla tra la parentesi che sta alla destra e il corpo del ciclo (alcuni linguaggi richiedono la parola do).

Quando l'istruzione while viene eseguita, per prima cosa viene analizzata l'espressione di controllo. Se il suo valore è diverso da zero (vero), il corpo del ciclo viene eseguito e l'espressione viene analizzata nuovamente. Il processo continua in questo modo (prima l'analisi dell'espressione di controllo e poi l'esecuzione del corpo del ciclo) fino a quando il valore dell'espressione di controllo non diventa uguale a zero.

L'esempio seguente utilizza l'istruzione while per calcolare la più piccola potenza di 2 che è maggiore o uguale al numero n:

```
i = 1;
while (i < n)
    i = i * 2;
```

Supponete che n abbia valore 10. La seguente traccia mostra cosa accade quando l'istruzione while viene eseguita:

```
i = 1;      adesso i vale 1.
i < n?      sì, continua.
i = i * 2   adesso i vale 2.
i < n?      sì, continua.
i = i * 2   adesso i vale 4.
i < n?      sì, continua.
i = i * 2   adesso i vale 8.
i < n?      sì, continua.
i = i * 2   adesso i vale 16.
i < n?      no, esci dal ciclo.
```

Osservate come il ciclo continui la sua esecuzione fintanto che l'espressione di controllo ($i < n$) è vera. Quando l'espressione diventerà falsa, il ciclo terminerà e si otterrà che, come desiderato, la variabile i avrà un valore maggiore o uguale a n.

Il fatto che il corpo del ciclo debba essere un'espressione singola non è altro che un mero dettaglio tecnico. Se vogliamo utilizzare più di un'istruzione, non dobbiamo far altro che inserire delle parentesi graffe in modo da creare un'istruzione composta [**istruzione composta > 5.2**]:

```
while (i > 0) {
    printf("T minus %d and counting\n", i);
    i--;
}
```

Alcuni programmati utilizzano sempre le parentesi graffe, anche quando non sono strettamente necessarie:

```
while (i < n) { /* parentesi graffe non necessarie ma ammesse */
    i = i * 2;
}
```

Come secondo esempio tracciamo l'esecuzione delle seguenti istruzioni che visualizzano una serie di messaggi di "conto alla rovescia":

```
i = 10;
while (i > 0) {
    printf("T minus %d and counting\n", i);
    i--;
}
```

Prima che il costrutto while venga eseguito, alla variabile i viene assegnato il valore 10. Dato che 10 è maggiore di 0, il corpo del ciclo viene eseguito comportando la stampa del messaggio T minus 10 and counting e il decremento della variabile i. Dato che 9 è maggiore di 0 il corpo del ciclo viene eseguito ancora una volta. Questo processo continua fino a quando non viene stampato il messaggio T minus 1 and counting e i diventa uguale a 0. Il test i > 0 fallisce causando la terminazione del ciclo.

L'esempio del conto alla rovescia ci conduce a diverse osservazioni riguardanti l'istruzione while.

- L'espressione di controllo è falsa quando il ciclo termina. Di conseguenza, quando un ciclo controllato dall'espressione $i > 0$ ha termine, i deve essere minore o uguale a 0 (se non fosse così staremmo ancora eseguendo il ciclo!).
- Il corpo del ciclo potrebbe non essere mai eseguito. Dato che l'espressione di controllo viene analizzata prima che il corpo del messaggio venga eseguito, è possibile che il corpo non venga eseguito nemmeno una volta. Se i avesse un valore negativo o uguale a zero al momento della prima entrata nel ciclo, quest'ultimo non farebbe nulla.
- Spesso un'istruzione while può essere scritta in diversi modi. Per esempio, avremmo potuto scrivere il ciclo del conto alla rovescia in una forma molto più concisa se avessimo decrementato i all'interno della printf:

 while (i > 0)
printf("T minus %d and counting\n", i--);

Cicli infiniti

Un'istruzione while non terminerà mai la sua esecuzione se l'espressione di controllo avrà sempre un valore diverso da zero. Spesso infatti i programmatore C creano deliberatamente un ciclo infinito utilizzando una costante diversa da zero come espressione di controllo:

```
while (1) ..
```

Un'istruzione while di questo tipo continuerà la sua esecuzione all'infinito a meno che il suo corpo non contenga un'istruzione in grado di trasferire il controllo fuori dal ciclo stesso (break, goto, return) o chiami una funzione che comporti la terminazione del programma.

PROGRAMMA Stampare la tavola dei quadrati

Scriviamo un programma che stampi la tavola dei quadrati. Il programma per p cosa chiederà all'utente di immettere un numero n . Successivamente verranno s pate n righe di output, ognuna delle quali contenente un numero compreso tra n assieme al suo quadrato:

This program prints a table of squares.
Enter number of entries in table: 5

```
1 1
2 4
3 9
4 16
5 25
```

Facciamo in modo che il programma memorizzi il numero dei quadrati in u variabile chiamata n . Avremo bisogno di un ciclo che stampi ripetutamente un n mero i e il suo quadrato, iniziando con i uguale a 1. Il ciclo si ripeterà fino a che non sarà minore o uguale a n . Dovremo anche assicurarci di incrementare i a og attraversamento del ciclo.

Scriveremo il ciclo con un istruzione `while` (francamente non abbiamo molta scel visto che `while` è l'unica istruzione che abbiamo trattato fino ad ora). Ecco il pro grammma finito:

```
square.c /* Stampa una tavola dei quadrati utilizzando l'istruzione while */
#include <stdio.h>

int main(void)
{
    int i, n;

    printf("This program prints a table of squares.\n");
    printf("Enter number of entries in table: ");
    scanf("%d", &n);

    i = 1;
    while (i <= n) {
        printf("%10d%10d\n", i, i * i);
        i++;
    }

    return 0;
}
```

Osservate che il programma `square.c` visualizza i numeri allineandoli perfettamente alle colonne. Il trucco è quello di utilizzare una specifica di conversione come `%10d` al posto della semplice `%d`. In questo modo si sfrutta il fatto che la `printf` allinea a destra i numeri nel caso in cui venga specificato un campo di larghezza per la stampa.

PROGRAMMA

Sommare una serie di numeri

Come secondo esempio dell'istruzione while possiamo scrivere un programma che somma una serie di interi immessi dall'utente. Ecco cosa vedrà l'utente:

This program sums a series of integers.

Enter integers (0 to terminate): 8 23 71 5 0

The sum is: 107

Chiaramente abbiamo bisogno di un ciclo che utilizzi la `scanf` per leggere e successivamente sommare i numeri al totale.

Dando alla variabile `n` il compito di rappresentare i numeri appena letti e a `sum` quello di memorizzare la somma dei numeri letti precedentemente, otteniamo il seguente programma:

```
sum.c /* Somma una sequenza di numeri */

#include <stdio.h>

int main(void)
{
    int n, sum = 0;

    printf("This program sums a series of integers.\n");
    printf("Enter integers (0 to terminate): ");

    scanf("%d", &n);
    while (n != 0) {
        sum += n;
        scanf("%d", &n);
    }
    printf("The sum is: %d\n", sum);

    return 0;
}
```

Osservate che la condizione `n != 0` viene testata solo dopo che un numero viene letto, permettendo così al ciclo di poter terminare il prima possibile. Osservate anche che ci sono due chiamate identiche alla `scanf`, il che è spesso difficile da evitare quando si utilizzano i cicli `while`.

6.2 L'istruzione do

L'istruzione `do` è strettamente collegata all'istruzione `while`, di fatto la sua essenza è quella di un'istruzione `while` nella quale l'espressione di controllo viene testata *dopo* l'esecuzione del corpo del ciclo. L'istruzione `do` ha la forma seguente:

do istruzione while (espressione)

Così come per l'istruzione `while`, il corpo di un'istruzione `do` deve essere composto da una sola istruzione (naturalmente sono ammesse anche le istruzioni composte) e l'espressione di controllo deve essere racchiusa tra parentesi.

Quando un'istruzione `do` viene eseguita, per prima cosa si esegue il corpo del ciclo e successivamente viene analizzata l'espressione di controllo. Se il valore dell'espressione non è uguale a zero, allora il corpo del ciclo viene eseguito ancora una volta e l'espressione di controllo viene analizzata nuovamente. L'esecuzione dell'istruzione `do` termina quando l'espressione di controllo ha valore 0 *successivamente* all'esecuzione del corpo del ciclo.

Riscriviamo l'esempio del conto alla rovescia della Sezione 6.1, utilizzando questa volta l'istruzione `do`:

```
i = 10;
do {
    printf("T minus %d and counting\n", i);
    --i;
} while (i > 0);
```

All'esecuzione dell'istruzione `do`, per prima cosa viene eseguito il corpo del ciclo facendo sì che il messaggio `T minus 10 and counting` venga visualizzato e che la variabile `i` venga decrementata. Successivamente viene controllata la condizione `i > 0`. Siccome 9 è maggiore di 0 il corpo del ciclo viene eseguito una seconda volta. Questo processo continua fino a quando viene visualizzato il messaggio `T minus 1 and counting` e la variabile `i` diventa 0. Questa volta il test `i > 0` fallisce causando il termine del ciclo. Come dimostra questo esempio, l'istruzione `do` è spesso indistinguibile dall'istruzione `while`. La differenza tra le due è che il corpo di un'istruzione `do` viene sempre eseguito almeno una volta, mentre il corpo dell'istruzione `while` viene interamente saltato se l'espressione di controllo è inizializzata a 0.

Si può affermare che è una buona pratica utilizzare le parentesi graffe in *tutte* le istruzioni `do`, sia che queste siano necessarie o meno. Il motivo è che un'istruzione `do` senza parentesi graffe potrebbe essere facilmente scambiata per un'istruzione `while`:

```
do
    printf("T minus %d and counting\n", i--);
while (i > 0);
```

Un lettore distratto potrebbe pensare che la parola `while` sia l'inizio di un costrutto `while`.

PROGRAMMA

Calcolare il numero di cifre in un intero

Sebbene l'istruzione `while` appaia nei programmi C con una frequenza maggiore rispetto all'istruzione `do`, quest'ultima è molto utile per i cicli che devono essere eseguiti almeno una volta. Per illustrare questo concetto, scriviamo un programma che calcoli il numero di cifre presenti in un intero immesso dall'utente:

```
Enter a nonnegative integer: 60
The number has 2 digit(s).
```

La nostra strategia sarà quella di dividere ripetutamente per 10 il numero immesso dall'utente fino a quando questo non diventa uguale a 0. Il numero di divisioni effettuate corrisponde al numero di cifre. Chiaramente avremo bisogno di un ciclo di qualche tipo dato che non sappiamo a priori quante divisioni saranno necessarie per

raggiungere lo 0. Dobbiamo usare l'istruzione `while` o l'istruzione `do?` L'istruzione `do` finisce per avere maggiore attrattiva dato che tutti gli interi (anche lo 0) hanno almeno una cifra. Ecco il programma:

```
numdigits.c /* Calcola il numero di cifre di un numero intero */
#include <stdio.h>
int main(void)
{
    int digits = 0, n;
    printf("Enter a nonnegative integer: ");
    scanf("%d", &n);

    do {
        n /= 10;
        digits++;
    } while (n > 0);

    printf("The number has %d digit(s).\n", digits);
    return 0;
}
```

Per capire perché l'istruzione `do` rappresenta la scelta corretta, vediamo cosa succederebbe se sostituissimo il ciclo `do` con un ciclo `while` simile:

```
while (n > 0) {
    n /= 10;
    digits++;
}
```

Se il valore iniziale di `n` fosse pari a 0, questo ciclo non verrebbe eseguito e il programma stamperebbe il messaggio

The number has 0 digit(s).

6.3 L'istruzione `for`

Trattiamo ora l'ultima delle istruzioni del C per i cicli: l'istruzione `for`. Non scorriatevi di fronte all'apparente complessità dell'istruzione `for`: agli effetti pratici è il modo migliore per scrivere molti cicli. L'istruzione `for` è ideale per quei cicli che hanno una variabile contatore, ciononostante è abbastanza versatile da essere utilizzabile anche per cicli di altro tipo.

L'istruzione `for` ha la seguente forma:

for (<i>espr1</i> , <i>espr2</i> , <i>espr3</i>) <i>istruzione</i>
--

Dove *espr1*, *espr2* ed *espr3* sono delle espressioni. Ecco un esempio:

```
for (i = 10; i > 0; i--)
    printf("T minus %d and counting\n", i);
```

Quando questa istruzione for viene eseguita la variabile i viene inizializzata a 10 e successivamente viene analizzata per controllare se è maggiore di 0. Dato che i è effettivamente maggiore di 0, allora viene visualizzato il messaggio T minus 10 and counting e la variabile viene decrementata. La condizione i > 0 viene poi controllata nuovamente. Il corpo del ciclo viene eseguito 10 volte in tutto con i che va da 10 fino a 1.

TEXR

L'istruzione for è strettamente collegata con l'istruzione while. Infatti, eccetto per alcuni rari casi, un ciclo for può essere sempre rimpiazzato da un ciclo while equivalente:

```
espr1;
while ( espr2 ) {
    istruzione
    espr3;
}
```

Come possiamo notare dallo schema appena presentato, *espr1* è un passaggio di inizializzazione che viene eseguito solamente una volta, prima dell'inizio del ciclo. *espr2* controlla la fine del ciclo (il ciclo continua fino a che il valore di *espr2* diventa diverso da zero). *espr3* è un'operazione che viene eseguita alla fine di ogni iterazione. Applicando questo schema all'esempio precedente per l'istruzione for otteniamo:

```
i = 10;
while (i > 0) {
    printf("T minus %d and counting\n", i);
    i--;
}
```

Studiare il costrutto while equivalente può aiutare a comprendere i punti più delicate dei cicli for. Supponiamo per esempio di rimpiazzare *i--* con *--i* nel ciclo for in esame:

```
for (i = 10; i > 0; --i)
    printf("T minus %d and counting\n", i);
```

Che effetti ha sul ciclo questa sostituzione? Analizzando il ciclo while equivalente vediamo che non vi è alcun effetto:

```
i = 10;
while (i > 0) {
    printf("T minus %d and counting\n", i);
    --i;
}
```

Dato che la prima e la terza espressione dell'istruzione for vengono eseguite come istruzioni a se stanti, il loro valore è irrilevante (sono utili solo per i loro side effect). Di conseguenza queste due espressioni di solito sono assegnamenti o espressioni di incremento/decremento.

Idiomi per l'istruzione for

Solitamente l'istruzione for è la scelta migliore per i cicli basati su conteggi di incremento o decremento di una variabile. Un ciclo for che conta per un totale di n volte ha la seguente forma:

- **Conteggio da 0 fino a n-1:**
for (*i* = 0; *i* < n; *i*++) ..
- **Conteggio da 0 fino a n:**
for (*i* = 0; *i* <= n; *i*++) ..
- **Conteggio da n-1 fino a 0:**
for (*i* = n; *i* >= 0; *i*++) ..
- **Conteggio da n fino a 1:**
for (*i* = n; *i* > 0; *i*++) ..

Seguire questi schemi vi aiuterà a evitare errori piuttosto comuni tra i programmati principianti:

- Usare < al posto di > (o viceversa) nelle espressioni di controllo. Osservate che i cicli che contano "all'insù" utilizzano gli operatori < o <=, mentre i cicli che contano "all'ingiù" si affidano a > o >=.
- Usare == nelle espressioni di controllo al posto di <, <=, > o >=. È necessario che l'espressione di controllo sia vera all'inizio del ciclo e che diventi falsa successivamente, quando questo deve terminare. Un test come *i* == n non ha molto senso perché non è vero all'inizio del ciclo.
- Gli errori di "off-by-one" causati per esempio dalla scrittura di *i* <= n al posto che *i* < n nell'espressione di controllo.

Omettere le espressioni nelle istruzioni for

L'istruzione for può essere anche più flessibile di quanto visto finora. Alcuni cicli for potrebbero non aver bisogno di tutte e tre le espressioni che vengono utilizzate normalmente. Per questo motivo il C ci permette l'omissione di alcune o persino di tutte le espressioni.

Se viene omessa la *prima* espressione, non viene eseguita nessuna inizializzazione prima dell'inizio del ciclo:

```
i = 10;
for (; i > 0; --i)
    printf("T minus %d and counting\n", i);
```

In questo esempio *i* è stata inizializzata con un'istruzione separata e così abbiamo omesso la prima espressione del costrutto for (notate che il punto e virgola tra la prima e la seconda espressione è rimasto. I due caratteri di punto e virgola devono essere sempre presenti anche quando abbiamo omesso qualche espressione).

Se omettiamo la *terza* delle espressioni allora il corpo del ciclo diventa responsabile nell'assicurare che il valore della seconda espressione possa, eventualmente, diventare falso. Il nostro esempio di istruzione for può diventare come il seguente:

```
for (i = 10; i > 0;)
    printf("T minus %d and counting\n", i--);
```

Per compensare l'omissione della terza espressione abbiamo sistemato il decremento della variabile *i* all'interno del corpo del ciclo.

Quando la prima e la terza espressione vengono omesse entrambe, allora il ciclo *for* non è altro che un'istruzione *while* sotto mentite spoglie. Per esempio, il ciclo

```
for (; i > 0;)
    printf("T minus %d and counting\n", i--);
```

è equivalente a

```
while (i > 0)
    printf("T minus %d and counting\n", i--);
```

La versione con il *while* è più chiara e comprensibile e di conseguenza deve essere preferita.

Se viene omessa la seconda espressione, allora questa viene considerata vera per default e quindi il ciclo *for* non ha termine (a meno che non venga fermato in altri modi).

D&R Alcuni programmatore utilizzano la seguente istruzione *for* per creare cicli infiniti:

```
for (;;) -
```

I cicli *for* nel C99



Nel C99 la prima espressione del *for* può essere rimpiazzata da una dichiarazione. Questa caratteristica permette ai programmatore di dichiarare una variabile da utilizzare all'interno del ciclo:

```
for (int i = 0; i < n; i++)
```

-

La variabile *i* dell'esempio non ha bisogno di essere dichiarata prima del ciclo (agli effetti pratici se la variabile *i* esistesse già, questa istruzione creerebbe una *nuova* versione di *i* che verrebbe utilizzata solamente all'interno del ciclo).

Una variabile dichiarata da un'istruzione *for* non è accessibile al di fuori del corpo del ciclo (diremo che non è *visibile* fuori dal ciclo):

```
for (int i = 0; i < n; i++) {
```

-

```
    printf("%d", i); /* corretto, i è visibile all'interno del ciclo */
```

-

```
}
```

printf("%d", i); **/* SBAGLIATO */**

È buona prassi far sì che le istruzioni *for* dichiarino le proprie variabili di controllo: è comodo e rende più facile la comprensione dei programmi. Tuttavia, se il programma avesse bisogno di accedere alla variabile dopo il termine del ciclo, allora sarebbe necessario utilizzare il vecchio formato per l'istruzione *for*.

Tra l'altro è ammessa la dichiarazione di più variabili, a patto che queste siano tutte dello stesso tipo:

```
for (int i = 0, j = 0; i < n; i++)
```

-

L'operatore virgola

Occasionalmente potremmo voler scrivere cicli for con due (o più) espressioni di inizializzazione, oppure che incrementino diverse variabili a ogni iterazione. Possiamo fare tutto questo utilizzando un'espressione con la virgola (*comma expression*) al posto della prima o terza espressione del costrutto for.

Una *comma expression* ha la forma



dove *expr1* ed *expr2* sono due espressioni qualsiasi. Una *comma expression* viene calcolata in due fasi: nella prima viene calcolata l'espressione *expr1*, il cui valore viene ignorato; nella seconda fase viene calcolata l'espressione *expr2*, il cui valore diventa quello dell'intera *comma expression*. Il calcolo di *expr1* deve avere sempre un side effect, altrimenti non ha alcuno scopo.

Supponiamo per esempio che i e j abbiano rispettivamente i valori 1 e 5. Quando la *comma expression* `++i, i + j` viene calcolata, i viene incrementata e poi avviene il calcolo di `i + j`. Risulta quindi che il valore dell'intera espressione è 7 (e, naturalmente, i finisce per avere il valore 2). L'ordine di precedenza dell'operatore virgola è minore rispetto a quello di tutti gli altri operatori, quindi non c'è alcun bisogno di mettere delle parentesi attorno a `++i` e a `i + j`.

In alcuni casi può essere necessario concatenare una serie di *comma expression*, così come a volte raggruppiamo delle assegnazioni. L'operatore virgola è associativo a sinistra e quindi

`i = 1, j = 2, k = i + j`

verrà interpretato dal compilatore come

`((i = 1), (j = 2)), (k = (i + j))`

Considerato che l'operando sinistro di una *comma expression* viene calcolato prima di quello destro, le assegnazioni `i = 1, j = 2, e k = i + j` vengono eseguite in ordine da sinistra a destra.

L'operatore virgola è pensato per quelle situazioni in cui il C richiede una singola espressione, ma potrebbero esserne necessarie due o più. In altre parole possiamo dire che la virgola ci permette di "incollare" assieme due espressioni al fine di ottenere un'espressione unica (notate la somiglianza con l'espressione composta che permette di trattare un gruppo di istruzioni come se fossero un'istruzione unica).

La necessità di "incollare" delle espressioni non si ritrova molto spesso. Come vedremo più avanti, certe definizioni di macro possono sfruttare l'operatore virgola [**definizioni di macro > 14.3**]. L'istruzione for è l'unica altra situazione dove è più probabile che si utilizzi l'operatore virgola. Supponete, per esempio, di voler inizializzare due variabili all'ingresso di un ciclo for. Invece di scrivere

```

sum = 0;
for (i = 1; i <= N; i++)
    sum += i;

```

potremmo scrivere

```
for (sum = 0, i = 1; i <= N; i++)
    sum += i;
```

L'espressione `sum = 0, i = 1` per prima cosa assegna 0 a `sum` e successivamente assegna 1 a `i`. Aggiungendo altre virgolette l'istruzione `for` sarebbe in grado di inizializzare più di due variabili.

MIGLIORAMENTO

Stampare la tavola dei quadrati (rivisitato)

Il programma `square.c` (Sezione 6.1) può essere migliorato convertendo il suo ciclo `while` in un ciclo `for`:

```
/* Stampa una tavola dei quadrati usando un ciclo for */

#include <stdio.h>

int main(void)
{
    int i, n;

    printf("This program prints a table of squares.\n");
    printf("Enter number of entries in table: ");
    scanf("%d", &n);

    for (i = 1; i <= n; i++)
        printf("%10d%10d\n", i, i * i);

    return 0;
}
```

Possiamo usare questo programma per illustrare un punto molto importante riguardante l'istruzione `for`: il C non impone alcuna restrizione sulle tre espressioni che controllano il suo comportamento. Nonostante queste espressioni vengano solitamente usate per inizializzare, analizzare e aggiornare la stessa variabile, non c'è nessuna necessità che siano in relazione una con l'altra. Considerate la seguente versione del programma:

```
/* Stampa una tavola dei quadrati usando un metodo strano */

#include <stdio.h>

int main(void)
{
    int i, n, odd, square;

    printf("This program prints a table of squares.\n");
    printf("Enter number of entries in table: ");
    scanf("%d", &n);

    i = 1;
    odd = 3;
    for (square = 1; i <= n; odd += 2) {
```

```

    printf("%10d%10d\n", i, square);
    ++i;
    square += odd;
}
return 0;
}

```

L'istruzione `for` di questo programma inizializza una variabile (`square`), ne analizza un'altra (`i`) e ne incrementa una terza (`odd`). La variabile `i` è il numero che deve essere elevato al quadrato, `square` è il quadrato di `i` e `odd` è il numero che deve essere sommato al quadrato corrente per ottenere il successivo (permettendo così al programma di calcolare i quadrati consecutivi senza eseguire nessuna moltiplicazione).

L'enorme flessibilità dell'istruzione `for` può risultare particolarmente utile in alcuni casi: vedremo che sarà di grande aiuto quando lavoreremo con le *liste linkate* [linked list > 17.5]. Tuttavia l'istruzione `for` può essere facilmente usata in modo non appropriato e quindi non abusatene. Il ciclo `for` presente in `square3.c` sarebbe stato molto più chiaro se avessimo sistemato il codice in modo da rendere esplicito il controllo da parte di `i`.

6.4 Uscire da un ciclo

Abbiamo visto come scrivere dei cicli che hanno un punto di uscita precedente al corpo del ciclo (usando le istruzioni `while` e `for`) oppure immediatamente dopo (usando l'istruzione `do`). In certi casi, però, avremo bisogno di un punto di uscita all'interno del ciclo e potremmo persino volere un ciclo con più punti di uscita. L'istruzione `break` rende possibile la scrittura di entrambi i tipi di cicli.

Dopo aver esaminato l'istruzione `break` daremo un'occhiata a una coppia di istruzioni imparentate con essa: `continue` e `goto`. L'istruzione `continue` permette di saltare una parte di iterazione senza per questo uscire dal ciclo. L'istruzione `goto` invece, permette al programma di saltare da un'istruzione a un'altra. In realtà, grazie alla disponibilità di istruzioni come `break` e `continue`, l'istruzione `goto` viene usata molto di rado.

L'istruzione `break`

Abbiamo già discusso di come l'istruzione `break` permetta di trasferire il controllo al di fuori di un costrutto `switch`. L'istruzione `break` può essere usata anche per uscire dai cicli `while`, `do` o `for`.

Supponete di scrivere un programma che controlli se il numero `n` è primo. Il nostro piano sarebbe quello di scrivere un ciclo `for` che divida il numero `n` per tutti i numeri compresi tra 2 ed `n-1`. Dobbiamo uscire dal ciclo non appena troviamo un divisore, in tal caso non ci sarebbe alcun motivo per continuare con le iterazioni rimanenti. Successivamente al termine del ciclo possiamo utilizzare un'istruzione `if` per determinare se la fine del ciclo è stata prematura (e quindi `n` non è primo) oppure normale (`n` è primo):

```

for (d = 2; d < n; d++)
    if (n % d == 0)
        break;

```

```

if (d < n)
    printf("%d is divisible by %d\n", n, d);
else
    printf("%d is prime\n", n);

```

L'istruzione break è particolarmente utile per scrivere quei cicli dove il punto d'uscita si trova in mezzo al corpo del ciclo, piuttosto che all'inizio o alla fine. Per esempio, cadono in questa categoria i cicli che leggono l'input dell'utente e che devono terminare quando viene immesso un particolare valore:

```

for(;;) {
    printf("Enter a number (enter 0 to stop): ");
    scanf("%d", &n);
    if (n == 0)
        break;
    printf("%d cubed is %d\n", n, n * n * n);
}

```

L'istruzione break trasferisce il controllo al di fuori della *più interna* istruzione while, do, for o switch. Quindi quando queste istruzioni vengono annidate, l'istruzione break può eludere solo un livello di annidamento. Prendete in considerazione il caso di un'istruzione switch annidata dentro un ciclo while:

```

while (...) {
    switch (...) {
        ...
        break;
        ...
    }
}

```

L'istruzione break trasferisce il controllo fuori dell'istruzione switch ma non fuori del ciclo while. Ritorneremo su questo punto più avanti.

L'istruzione continue

L'istruzione continue non fuoriesce da un ciclo. Tuttavia, data la sua somiglianza con l'istruzione break, l'inclusione in questa sezione non è del tutto arbitraria. L'istruzione break trasferisce il controllo in un punto immediatamente *successivo* alla fine del ciclo, mentre l'istruzione continue trasferisce il controllo a un punto immediatamente *precedente* al corpo del ciclo. Con break il controllo fuoriesce dal ciclo, con continue il controllo rimane all'interno del ciclo. Un'altra differenza tra le due istruzioni è che break può essere usata sia nei costrutti switch che nei cicli (while, do e for), mentre continue ha un utilizzo limitato solamente ai cicli.

L'esempio seguente, che legge una serie di numeri e calcola la loro somma, illustra un semplice utilizzo dell'istruzione continue. Il ciclo termina quando sono stati letti 10 numeri diversi da zero. Ogni volta che viene letto un numero uguale a zero, viene eseguita l'istruzione continue che salta la parte restante del corpo del ciclo (le istruzioni `sum += i;` e `n++;`) rimanendo comunque all'interno di quest'ultimo.

```

n = 0;
sum = 0;
while (n < 10) {
    scanf("%d", &i);
    if (i == 0)
        continue;
    sum += i;
    n++;
    /* continue salta qui */
}

```

Se continue non fosse stata disponibile avremmo scritto l'esempio in questo modo:

```

n = 0;
sum = 0;
while (n < 10) {
    scanf("%d", &i);
    if (i != 0) {
        sum += i;
        n++;
    }
}

```

L'istruzione goto

Sia break che continue sono istruzioni di salto che trasferiscono il controllo da un punto del programma a un altro. Sono entrambe limitate: l'obiettivo del break è un punto immediatamente *successivo* alla fine del ciclo, mentre l'obiettivo di un'istruzione continue è un punto che si trova immediatamente *prima* la fine del ciclo. L'istruzione goto, invece, è in grado di saltare verso una qualsiasi istruzione contenuta all'interno di una funzione, ammesso che questa istruzione sia provvista di una **label** (etichetta) (il C99 impone un'ulteriore restrizione alla goto: non può essere usata per bypassare la dichiarazione di un vettore a dimensione variabile [**vettori a dimensione variabile > 8.3**]).

C99

Una label non è altro che un identificatore messo all'inizio di un'istruzione:

identificatore : istruzione

Un'istruzione può avere più di una label. L'istruzione goto ha il seguente formato

goto identificatore;

Eseguire l'istruzione goto *L*; trasferisce il controllo all'istruzione che segue la label *L*, la quale deve essere all'interno della stessa funzione in cui si trova l'istruzione goto.

Se il C non avesse avuto l'istruzione break, ecco come avremmo potuto usare goto per uscire prematuramente da un ciclo:

```

for (d = 2; d < n; d++)
    if (n % d == 0)
        goto done;

done:
if (d < n)
    printf("%d is divisible by %d\n", n, d);
else
    printf("%d is prime\n", n);

```



La `goto`, che era una delle istruzioni principali nei vecchi linguaggi di programmazione, viene usata raramente nella programmazione C attuale. Le istruzioni `break`, `continue` e `return` (che sono essenzialmente delle istruzioni `goto` limitate) e la funzione `exit` [funzione `exit` > 9.5] sono sufficienti per gestire la maggior parte delle situazioni dove in altri linguaggi di programmazione è necessaria l'istruzione `goto`.

Detto questo, a volte l'istruzione `goto` può essere pratica da utilizzare. Considerate il problema di uscire da un ciclo dall'interno di una struttura `switch`. Come abbiamo visto precedentemente, l'istruzione `break` non porta all'effetto desiderato: esce dalla struttura `switch` ma non dal ciclo. Un'istruzione `goto` risolve il problema:

```

while (...) {
    switch (...) {
        -
        goto loop_done; /* l'istruzione break non funzionerebbe qui */
        -
    }
}
loop_done: -

```

L'istruzione `goto` è utile anche per uscire dai cicli annidati.

PROGRAMMA

Bilancio di un conto

Tanti semplici programmi interattivi sono basati su menu: presentano all'utente una lista di possibili comandi tra cui scegliere. Una volta che l'utente ha selezionato un comando, il programma esegue l'azione desiderata e chiede all'utente l'immissione di un comando nuovo. Questo procedimento continua fino a che l'utente non seleziona un comando come `exit` o `quit`.

Ovviamente il cuore di un programma di questo tipo è un ciclo. All'interno del ciclo ci saranno delle istruzioni che chiedono all'utente un comando, lo leggono e poi decidono che azione intraprendere:

```

for (;;) {
    chiede all'utente il comando;
    legge il comando;
    esegue il comando;
}

```

L'esecuzione del comando richiederà una struttura switch (o una serie di if in cascata):

```
for (;;) {
    chiede all'utente il comando;
    legge il comando;
    switch (comando) {
        case comando1 : esegui operazione1; break;
        case comando2 : esegui operazione2; break;
        .
        .
        .
        case comandon : esegui operazionen; break;
        default: stampa messaggio di errore; break;
    }
}
```

Per illustrare questa struttura sviluppiamo un programma che mantenga il bilancio di un conto. Il programma presenterà all'utente una serie di scelte: azzerare il conto, accreditare o addebitare denaro sul conto, stampare l'attuale situazione del conto, uscire dal programma. Le scelte vengono rispettivamente rappresentate dagli interi 0, 1, 2, 3 e 4. Ecco come dovrebbe apparire una sessione con questo programma:

```
*** ACME checkbook-balancing program ***
Commands: 0=clear, 1=credit, 2=debit, 3=balance, 4=exit
Enter command: 1
Enter amount of credit: 1042.56
Enter command: 2
Enter amount of debit: 133.79
Enter command: 1
Enter amount of credit: 1754.32
Enter command: 2
Enter amount of debit: 1400
Enter command: 2
Enter amount of debit: 68
Enter command: 2
Enter amount of debit: 50
Enter command: 3
Current balance: $1145.09
Enter command: 4
```

Quando l'utente immette il comando 4 (exit) il programma ha bisogno di uscire dalla struttura switch e dal ciclo che la circonda. L'istruzione break non sarà di aiuto e per questo sarebbe preferibile l'istruzione goto. Tuttavia nel programma useremo l'istruzione return che imporrà alla funzione main di ritornare il controllo al sistema operativo.

```

checking.c /* Bilancio di un conto */

#include <stdio.h>

int main(void)
{
    int cmd;
    float balance = 0.0f, credit, debit;

    printf("/** ACME checkbook-balancing program **\\n");
    printf("Commands: 0=clear, 1=credit, 2=debit, ");
    printf("3=balance, 4=exit\\n\\n");
    for (;;) {
        printf("Enter command: ");
        scanf("%d", &cmd);
        switch (cmd) {
            case 0:
                balance = 0.0f;
                break;
            case 1:
                printf("Enter amount of credit: ");
                scanf("%f", &credit);
                balance += credit;
                break;
            case 2:
                printf("Enter amount of debit: ");
                scanf("%f", &debit);
                balance -= debit;
                break;
            case 3:
                printf("Current balance: $%.2f\\n", balance);
                break;
            case 4:
                return 0;
            default:
                printf("Commands: 0=clear, 1=credit, 2=debit, ");
                printf("3=balance, 4=exit\\n\\n");
                break;
        }
    }
}

```

Osservate che l'istruzione `return` non è seguita dall'istruzione `break`. Un `break` chiuso si trovi immediatamente dopo un `return` non potrà mai essere eseguito, per questo motivo molti compilatori generano un messaggio di errore.

6.5 L'istruzione vuota

Un'istruzione potrebbe essere **vuota** ovvero sprovvista di qualsiasi simbolo fatta eccezione per il punto e virgola alla fine. Ecco un esempio:

```
i = 0; ; j = 1;
```

Questa riga contiene tre istruzioni: un'assegnazione a i, un'istruzione vuota e un'assegnazione a j.

D&R

L'istruzione vuota (*null statement*) è utile per scrivere cicli il cui corpo è vuoto. Per fare un esempio richiamiamo il ciclo presentato nella Sezione 6.4 per la ricerca di numeri primi:

```
for (d = 2; d < n; d++)
    if (n % d == 0)
        break;
```

Se spostiamo la condizione $n \% d == 0$ all'interno dell'espressione di controllo del ciclo il corpo del ciclo stesso diventa vuoto:

```
for (d = 2; d < n && n % d != 0; d++)
/* ciclo con corpo vuoto */;
```

Ogni volta che il ciclo viene attraversato, per prima cosa viene controllata la condizione $d < n$. Se questa è falsa il ciclo ha termine, altrimenti viene controllata la condizione $n \% d != 0$, la quale, se falsa, fa terminare il ciclo (in quel caso sarebbe vera la condizione $n \% d == 0$ e quindi avremmo trovato un divisore di n).

Prestate attenzione a come l'istruzione vuota sia stata messa in una riga a sé stante in luogo di scrivere

```
for (d = 2; d < n && n % d != 0; d++);
```

D&R

Per consuetudine i programmati C pongono le istruzioni vuote in una riga a sé stante. Se non si agisse in questo modo si potrebbe generare confusione nella lettura del programma facendo erroneamente pensare che l'istruzione successiva a quella del ciclo for faccia parte del corpo di quest'ultimo:

```
for (d = 2; d < n && n % d != 0; d++);
if (d < n)
    printf("%d is divisible by %d\n", n, d);
```

Non si guadagna molto convertendo un normale ciclo in un ciclo con corpo vuoto: il nuovo ciclo è più conciso ma tipicamente non è più efficiente. In certi casi però, un ciclo con corpo vuoto è nettamente migliore delle alternative. Per esempio, vedremo come questi cicli siano particolarmente comodi per leggere caratteri [leggere caratteri > 7.3].

Inserire accidentalmente un punto e virgola dopo le parentesi delle istruzioni if, while o for crea un'istruzione vuota che causa la fine prematura dell'istruzione.



- Inserire un punto e virgola dopo le parentesi di un'istruzione if crea apparentemente un if che esegue la stessa azione senza curarsi del valore dell'espressione controllo:

```
if (d == 0);
printf("Error: Division by zero\n");
```

/** SBAGLIATO **

La chiamata alla printf non fa parte dell'istruzione if e quindi viene eseguita indipendentemente dal valore della variabile d.

- In un'istruzione while, mettere un punto e virgola dopo le parentesi può creare un ciclo infinito:

```
i = 10;
while (i > 0);
{
    printf("T minus %d and counting\n", i);
    --i;
}
```

/** SBAGLIATO **

Un'altra possibilità è che il ciclo abbia termine e che l'istruzione che dovrebbe costituire il corpo venga eseguita solamente una volta dopo il termine del ciclo stesso.

```
i = 11;
while (-i > 0);
printf("T minus %d and counting\n", i);
```

/** SBAGLIATO **

Questo esempio visualizzerebbe il messaggio
T minus 0 and counting

- Mettere un punto e virgola subito dopo le parentesi di un'istruzione for porterebbe l'istruzione che forma il corpo del ciclo ad essere eseguita una sola volta.

```
for (i = 10; i > 0; i--);
printf("T minus %d and counting\n", i);
```

/** SBAGLIATO **

Anche questo esempio stampa il messaggio
T minus 0 and counting

Domande & Risposte

D: Il ciclo seguente appare nella Sezione 6.1

```
while (i > 0)
printf("T minus %d and counting\n", i);
```

Perché non abbreviamo ulteriormente il ciclo rimuovendo la scrittura “> 0”?

```
while (i)
printf("T minus %d and counting\n", i);
```

Questa versione si fermerebbe non appena i raggiunge lo 0 e quindi dovrebbe essere funzionante come l'originale. [p. 105]

R: La nuova versione è sicuramente più concisa e molti programmati C scriverebbero il ciclo in questo modo, tuttavia ci sono alcuni inconvenienti.

Per prima cosa il ciclo non è facilmente leggibile come l'originale. È chiaro che il ciclo abbia termine quando i raggiunge lo 0 ma non è chiaro se stiamo contando in avanti o all'indietro. Nel ciclo originale questa informazione può essere dedotta dall'espressione di controllo $i > 0$.

In secondo luogo, il nuovo ciclo si comporterebbe in modo differente nel caso in cui i avesse un valore negativo al momento in cui il ciclo stesso iniziasse l'esecuzione. Il ciclo originale terminerebbe subito, mentre non lo farebbe la nuova versione.

D: La Sezione 6.3 dice che i cicli for possono essere convertiti in cicli while utilizzando uno schema standard a eccezione di rari casi. Potrebbe fare un esempio di uno di questi casi? [p. 110]

R: Quando il corpo di un ciclo for contiene un'istruzione continue, lo schema visto nella Sezione 6.3 non è più valido. Considerate l'esempio seguente preso dalla Sezione 6.4:

```
n = 0;
sum = 0;
while (n < 10) {
    scanf("%d", &i);
    if (i == 0)
        continue;
    sum += i;
    n++;
}
```

A prima vista sembra possibile convertire il ciclo while in un ciclo for:

```
sum = 0;
for (n = 0; n < 10; n++) {
    scanf("%d", &i);
    if (i == 0)
        continue;
    sum += i;
}
```

Sfortunatamente questo ciclo non è equivalente all'originale. Quando i è uguale a 0 il ciclo originale non incrementa n, mentre questo è quello che avviene con il nuovo ciclo.

D: Quale forma di ciclo infinito è preferibile, while (1) o for (;;) ? [p. 112]

R: Tradizionalmente i programmati C utilizzano la forma for (;;) per ragioni di efficienza. I vecchi compilatori spesso forzavano i programmi a controllare la condizione 1 a ogni iterazione del ciclo while. Con i moderni compilatori però non ci sono differenze in termini di performance.

D: Abbiamo sentito che i programmatori non dovrebbero mai usare l'istruzione continue. È vero?

R: È vero che le istruzioni continue sono rare, tuttavia in certi casi sono comode da usare. Supponete di scrivere un ciclo che legga dei dati di input, controlli che questi siano validi e in tal caso li elabori in qualche modo. Se vi sono diversi test di validità o se questi sono complessi, l'istruzione continue può essere utile. Il ciclo apparirebbe in questo modo:

```
for (;;) {
    leggi il dato;
    if (il dato fallisce il primo test)
        continue;
    if (il dato fallisce il secondo test)
        continue;
    .
    .
    if (il dato fallisce l'ultimo test)
        continue;
    elabora i dati;
}
```

D: Perché l'istruzione goto va usata con parsimonia? [p. 118]

R: L'istruzione goto non è intrinsecamente "cattiva", ma vi sono alternative migliori. I programmi che usano più di una manciata di goto possono facilmente degenerare nel cosiddetto *spaghetti code*, dove il controllo salta spensieratamente da un punto all'altro del programma. I programmi spaghetti code sono difficili da capire e soprattutto difficili da modificare.

L'istruzione goto rende difficile la lettura dei programmi perché i salti possono essere sia in avanti che all'indietro (al contrario di break e continue che saltano solo in avanti). Un programma che contiene istruzioni goto richiede al lettore di saltare spesso in avanti e indietro nel tentativo di seguire il controllo del flusso.

L'istruzione goto può rendere i programmi difficili da modificare in quanto essi permettono a una sezione di codice di servire a più scopi. Un'istruzione preceduta da un'etichetta per esempio può essere raggiunta sia "scendendo" dall'istruzione precedente che eseguendo diverse istruzioni goto.

D: L'istruzione vuota possiede altri scopi oltre a quello di indicare che il corpo di un ciclo è vuoto? [p. 121]

R: Molto pochi. Considerato che l'istruzione vuota può trovarsi in ogni punto dove è ammessa un'istruzione, gli usi potenziali possono essere molti. Tuttavia nella pratica c'è solo un altro utilizzo dell'istruzione vuota, ed è raro.

Supponete di aver bisogno di una label alla fine di un'istruzione composta. Una label non può restare isolata, deve essere sempre seguita da un'istruzione. Mettendo un'istruzione vuota dopo la label si risolve il problema:

```
{
    - goto fine_dell_istr;
    -
    fine_dell_istr: ;
}
```

D: Ci sono altri modi di evidenziare un ciclo con il corpo vuoto oltre a quello di mettere un'istruzione vuota in una riga a sé stante? [p. 121]

R: Alcuni programmatore utilizzano un'istruzione continue inutile:

```
for (d = 2; d < n && n % d != 0; d++)
    continue;
altri usano un'istruzione composta vuota
for (d = 2; d < n && n % d != 0; d++)
{}
```

Esercizi

Sezione 6.1 1. Qual è l'output prodotto dal seguente frammento di programma?

```
i = 1;
while (i <= 128) {
    printf("%d ", i);
    i *= 2;
}
```

Sezione 6.2 2. Qual è l'output prodotto dal seguente frammento di programma?

```
i = 9384;
do {
    printf("%d ", i);
    i /= 10;
} while (i > 0);
```

Sezione 6.3 3. *Qual è l'output prodotto dal seguente frammento di programma?

```
for (i = 5, j = i - 1; i > 0, j > 0; --i, j = i - 1)
    printf("%d ", i);
```

- W 4. Quale delle seguenti istruzioni non è equivalente alle altre due (assumendo che il corpo del ciclo sia lo stesso per tutte)?
- for (i = 0; i < 10; i++) ..
 - for (i = 0; i < 10; ++i) ..
 - for (i = 0; i++ < 10;) ..

5. Quale delle seguenti istruzioni non è equivalente alle altre due (assumendo che il corpo del ciclo sia lo stesso per tutte)?
- while (*i* < 10) {}
 - for (; *i* < 10;) {}
 - do {} while (*i* < 10);
6. Traducete il frammento di programma dell'Esercizio 1 in una singola istruzione for.
7. Traducete il frammento di programma dell'Esercizio 2 in una singola istruzione for.
8. *Qual è l'output prodotto dal seguente frammento di programma?
- ```
for (i = 10; i >= 1; i /= 2)
 printf("%d ", i++);
```
9. Traducete l'istruzione for dell'Esercizio 8 in un ciclo while equivalente. Avete bisogno di un'altra istruzione in aggiunta alla while.

**Sezione 6.4** W 10. Mostrate come si sostituisce un'istruzione continue con un'istruzione goto.

11. Qual è l'output prodotto dal seguente frammento di programma?

```
sum = 0;
for (i = 0; i < 10; i++) {
 if (i % 2)
 continue;
 sum += i;
}
printf("%d\n", sum);
```

W 12. Il seguente ciclo per il test dei numeri primi è stato illustrato come esempio nella Sezione 6.4:

```
for (d = 2; d < n; d++)
 if (n % d == 0)
 break;
```

Questo ciclo non è molto efficiente. Per determinare se *n* è primo non è necessario dividerlo per tutti i numeri compresi tra 2 e *n*-1. Infatti abbiamo bisogno di cercare i divisori solamente fino alla radice quadrata di *n*. Modificate il codice per tenere conto di questo fatto. Suggerimento: non cercate di calcolare la radice quadrata di *n*, piuttosto fate il confronto tra *d* \* *d* ed *n*.

**Sezione 6.3** 13. \*Riscrivete il ciclo seguente in modo che il suo corpo sia vuoto:

```
for (n = 0; m > 0; n++)
 m /= 2;
```

W 14. \*Trovate l'errore presente nel seguente frammento di programma e correggetelo.

```
if (n % 2 == 0);
printf("n is even\n");
```

## Progetti di programmazione

- Scrivete un programma che, data una serie di numeri immessi dall'utente, ne trova il maggiore. Il programma deve chiedere all'utente di immettere i numeri uno alla volta. Quando l'utente immette un numero negativo o lo zero, il programma deve visualizzare il più grande numero non negativo immesso fino a quel momento:

Enter a number: 60  
 Enter a number: 38.3  
 Enter a number: 4.89  
 Enter a number: 100.62  
 Enter a number: 75.2295  
 Enter a number: 0

The largest number entered was 100.62

Tenete presente che i numeri non sono necessariamente interi.

- W 2. Scrivete un programma che chieda all'utente di immettere due interi e poi calcoli e visualizzi il loro massimo comun divisore (MCD):

Enter two integers: 12 28  
 Greatest common divisor: 4

*Suggerimento:* l'algoritmo classico per il calcolo dell'MCD, conosciuto come algoritmo di Euclide, agisce in questo modo: siano  $m$  ed  $n$  le variabili contenenti i due numeri. Assumendo che  $m$  sia maggiore di  $n$ , se  $n$  è uguale a 0 allora ci si ferma perché  $m$  contiene il MCD. Altrimenti calcola il resto della divisione tra  $m$  ed  $n$ . Si deve copiare il contenuto di  $n$  in  $m$  e copiare il resto ottenuto dalla divisione in  $n$ . Il procedimento va ripetuto, verificando se  $n$  è uguale a 0.

- Scrivete un programma che chieda all'utente di immettere una frazione e successivamente riduca quella frazione ai minimi termini:

Enter a fraction: 6/12  
 In lowest terms: 1/2

*Suggerimento:* per ridurre una frazione ai minimi termini, per prima cosa calcolate il MCD del numeratore e del denominatore. Successivamente dividete sia il numeratore che il denominatore per il MCD.

- W 4. Aggiungete un ciclo al programma broker.c della Sezione 5.2 in modo che l'utente possa immettere più di uno scambio e il programma calcoli la commissione su ognuno di questi. Il programma deve terminare quando l'utente immette 0 come valore dello scambio:

Enter value of trade: 30000  
 Commission: \$166.00  
 Enter value of trade: 20000  
 Commission: \$144.00  
 Enter value of trade: 0

5. Il Progetto di programmazione 1 del Capitolo 4 vi ha chiesto di scrivere un programma che visualizzi un numero a due cifre invertendo l'ordine di queste ultime. Generalizzate il programma in modo che il numero possa avere una, due, tre o più cifre. *Suggerimento:* usare un ciclo `do` che divide ripetutamente il numero per 10 fermandosi al raggiungimento dello 0.
- W 6. Scrivete un programma che chieda all'utente di immettere un numero  $n$  e successivamente stampi tutti i quadrati pari compresi tra 1 ed  $n$ . Per esempio, se l'utente immettesse 100, il programma dovrebbe stampare il seguente risultato:

```
4
16
36
64
100
```

7. Sistematate il programma `square3.c` in modo che il ciclo `for` inizializzi, controlli e incrementi la variabile `i`. Non riscrivete il programma, e in particolare non usate nessuna moltiplicazione.
- W 8. Scrivete un programma che stampi il calendario di un mese. L'utente deve specificare il numero di giorni nel mese e il giorno della settimana in cui questo comincia:

`Enter number of days in month: 31`

`Enter starting day of the week (1=Sun, 7=Sat): 3`

|    | 1  | 2  | 3  | 4  | 5  |
|----|----|----|----|----|----|
| 6  | 7  | 8  | 9  | 10 | 11 |
| 13 | 14 | 15 | 16 | 17 | 18 |
| 20 | 21 | 22 | 23 | 24 | 25 |
| 27 | 28 | 29 | 30 | 31 |    |

*Suggerimento:* questo programma non è difficile come sembra. La parte più importante è il ciclo `for` che usa la variabile `i` per contare da 1 a  $n$  (dove  $n$  è il numero di giorni del mese) e stampa tutti i valori di `i`. All'interno del ciclo un'istruzione `if` controlla se `i` è l'ultimo giorno della settimana e in quel caso stampa un carattere new-line.

9. Nel Progetto di programmazione 8 del Capitolo 2 veniva chiesto di scrivere un programma che calcolasse il debito residuo di un prestito dopo la prima, la seconda e la terza rata mensile. Modificate il programma in modo che chieda all'utente di inserire anche il numero di pagamenti e successivamente visualizzi il debito residuo dopo ognuno di questi pagamenti.
10. Nel Progetto di programmazione del Capitolo 5 è stato chiesto di scrivere un programma che determinasse quale delle due date venisse prima nel calendario. Generalizzate il programma in modo che l'utente possa immettere un numero qualsiasi di date. L'utente dovrà immettere 0/0/0 per segnalare che non immetterà ulteriori date:

Enter a date (mm/dd/yy): 3/6/08

Enter a date (mm/dd/yy): 5/17/07

Enter a date (mm/dd/yy): 6/3/07

Enter a date (mm/dd/yy): 0/0/0

5/17/07 is the earliest date

11. Il valore della costante matematica  $e$  può essere espresso come una serie infinita:

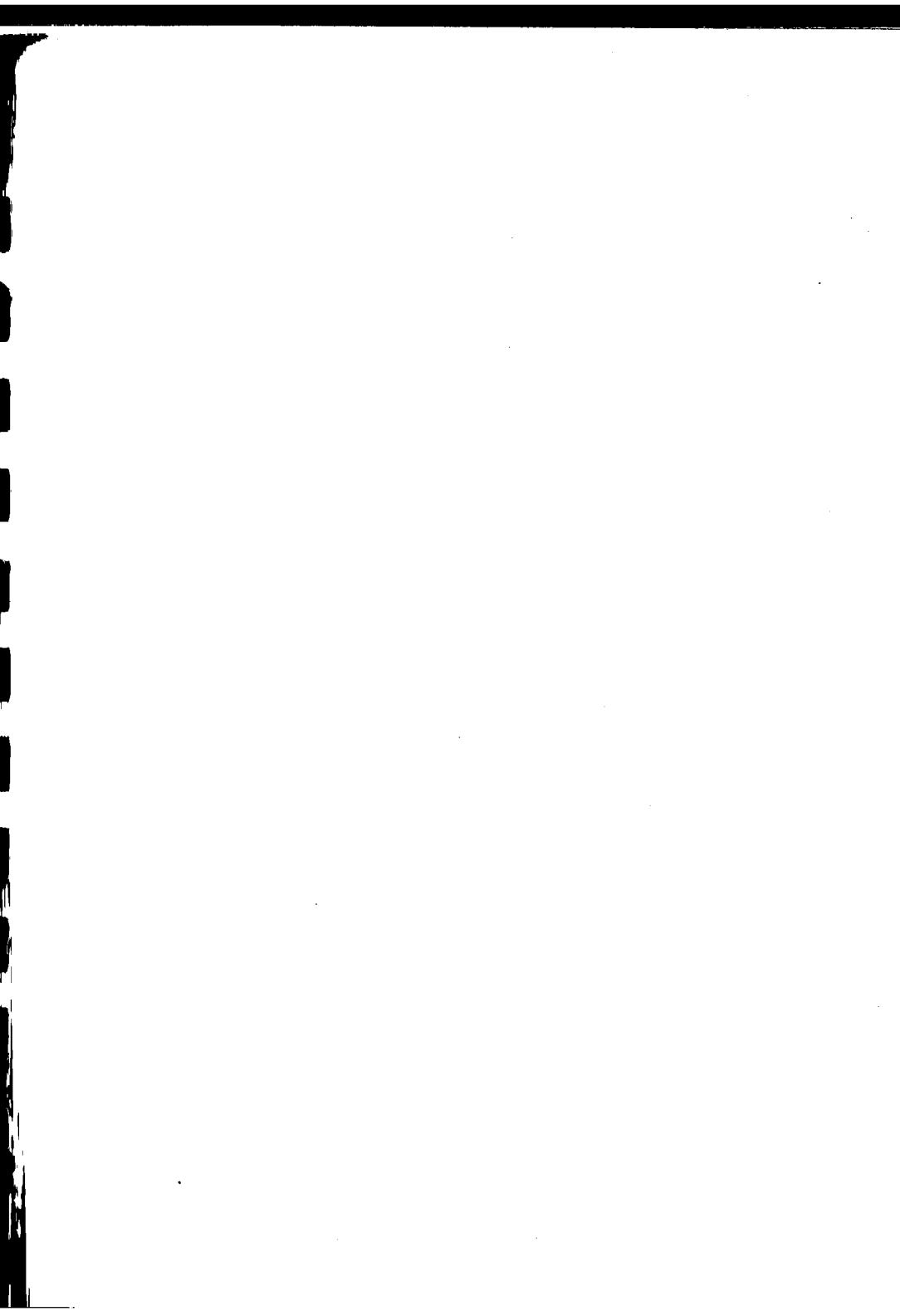
$$e = 1 + 1/1! + 1/2! + 1/3! + \dots$$

Scrivete un programma che approssimi  $e$  calcolando il valore di

$$1 + 1/1! + 1/2! + 1/3! + 1/n!$$

dove  $n$  è un intero immesso dall'utente.

12. Modificate il Progetto di programmazione 11 in modo che il programma continui a sommare termini fino a che il termine corrente non diventa inferiore a  $\epsilon$ , dove  $\epsilon$  è un piccolo numero (*floating point*) immesso dall'utente.



# 7 I tipi base

Finora abbiamo utilizzato solamente due **tipi base** del C: int e float (abbiamo dato anche un tipo base del C99 chiamato \_Bool). Questo capitolo descrive gli altri tipi base e tratta di questioni di una certa importanza riguardanti i tipi in generale. La Sezione 7.1 illustra l'assetto completo dei tipi interi, che include gli interi long, short e unsigned. La Sezione 7.2 introduce i tipi double e long double che permettono un range e una precisione più grandi rispetto ai float. La Sezione 7.3 tratta il tipo char, del quale avremo bisogno per lavorare con i caratteri. La Sezione 7.4 tratta il delicato argomento della conversione da un valore di un tipo a un valore equivalente di un altro tipo. La Sezione 7.5 illustra l'uso di typedef per la definizione di nuovi nomi per i tipi. Infine la Sezione 7.6 descrive l'operatore sizeof che misura lo spazio di memoria richiesto per un tipo.

## 7.1 Tipi interi

Il C supporta due tipologie fondamentali di numeri: i numeri interi e quelli a virgola mobile. I valori di un tipo **intero** sono numeri interi, mentre i valori dei tipi a virgola mobile possono avere anche una parte frazionaria. I tipi interi possono a loro volta essere suddivisi in due categorie: interi con segno (signed) e interi senza segno (unsigned).

### Interi signed e unsigned

Il bit più significativo di un intero di tipo signed (conosciuto come bit di segno) è uguale a 0 se il numero è positivo o uguale a zero. È uguale a 1 se il numero è negativo. Quindi il più grande intero a 16 bit ha la seguente rappresentazione binaria:

0111111111111111

che corrisponde a 32,767 ( $2^{15}-1$ ). L'intero a 32 bit più grande è

01111111111111111111111111111111

che corrisponde a 2,147,483,647 ( $2^{31}-1$ ). Un intero senza bit di segno (il bit più significativo è inteso come parte integrante del numero) viene detto unsigned. L'intero più grande senza segno su 16

bit è 65,535 ( $2^{16}-1$ ), mentre il più grande intero senza segno rappresentato su 32 bit è 4,294,967,831 ( $2^{32}-1$ ).

Per default le variabili intere del C sono di tipo `signed` (il bit più significativo è riservato al segno). Per istruire il compilatore in modo che una variabile non abbia il bit di segno, dobbiamo dichiararla `unsigned`. I numeri senza segno sono utili principalmente per la programmazione di sistemi e applicazioni a basso livello dipendenti dalla macchina. Discuteremo di applicazioni tipiche di numeri senza segno nel Capitolo 20, fino ad allora tenderemo a evitarli.

I tipi di numeri interi del C hanno diverse dimensioni. Il tipo `int` di solito è 16 bit, ma in alcune vecchie CPU capita che sia di 16 bit. Dato che alcuni programmi lavorano con numeri che sono troppo grandi per essere memorizzati in una variabile `int`, il C fornisce anche gli interi di tipo `long`. In certi casi invece potremmo aver bisogno di risparmiare la memoria disponibile e imporre al compilatore di riservare uno spazio inferiore al normale per la memorizzazione di un numero. In tal caso useremo una variabile di tipo `short`.

Per costruire un tipo intero che venga incontro alle nostre necessità, possiamo specificare una variabile come `long` o `short`, `signed` o `unsigned`. Possiamo anche combinarne questi specificatori (per esempio `long unsigned int`). In realtà nella pratica solo le seguenti combinazioni generano dei tipi differenti:

```
short int
unsigned short int

int
unsigned int

long int
unsigned long int
```

Le altre combinazioni costituiscono dei sinonimi per questi sei tipi (per esempio `signed int` equivale a `long int` dato che gli interi sono sempre con segno, a meno che non venga specificato diversamente). L'ordine degli specificatori non ha importanza, infatti `unsigned short int` equivale a `short unsigned int`.

Il C permette l'abbreviazione dei nomi per i numeri interi con l'omissione della parola `int`. Per esempio `unsigned short int` può essere abbreviato con `unsigned short`, mentre `long int` può essere abbreviato con il semplice `long`. L'omissione di `int` è una pratica molto diffusa tra i programmatore C, tanto che alcuni linguaggi recenti basati sul C (Java incluso) richiedono che il programmatore descriva `short` o `long` al posto di `short int` o `long int`. Per queste ragioni ometterò spesso la parola `int` quando non è strettamente necessaria.

L'intervallo dei valori rappresentabili con i sei tipi interi citati varia da una macchina all'altra. Ci sono tuttavia un paio di regole alle quali tutti i compilatori devono obbedire. Per prima cosa lo standard C richiede che `short int`, `int` e `long int` copriano un certo intervallo minimo di valori (guardate la Sezione 23.3 per i dettagli). Secondariamente lo standard richiede che il tipo `int` non sia più piccolo di `short int` e che il tipo `long int` non sia più piccolo di `int`. È possibile tuttavia che il tipo `short int` rappresenti lo stesso range di valori del tipo `int`.

La Tabella 7.1 illustra l'intervallo di valori solitamente associati ai tipi interi su una macchina a 16 bit. Ricordate che short int e int hanno intervalli identici.

**Tabella 7.1** I Tipi interi su una macchina a 16 bit

| Tipo               | Valore più piccolo | Valore più grande |
|--------------------|--------------------|-------------------|
| short int          | -32,768            | 32,767            |
| unsigned short int | 0                  | 65,535            |
| int                | -32,768            | 32,767            |
| unsigned int       | 0                  | 65,535            |
| long int           | -2,147,483,648     | 2,147,483,647     |
| unsigned long int  | 0                  | 4,294,697,295     |

La Tabella 7.2 illustra l'intervallo di valori su una macchina a 32 bit. Qui int e long int hanno intervalli identici.

**Tabella 7.2** I tipi interi su una macchina a 32 bit

| Tipo               | Valore più piccolo | Valore più grande |
|--------------------|--------------------|-------------------|
| short int          | -32,768            | 32,767            |
| unsigned short int | 0                  | 65,535            |
| int                | -2,147,483,648     | 2,147,483,647     |
| unsigned int       | 0                  | 4,294,697,295     |
| long int           | -2,147,483,648     | 2,147,483,647     |
| unsigned long int  | 0                  | 4,294,697,295     |

Negli ultimi anni le CPU a 64 bit sono diventate più comuni. La Tabella 7.3 illustra gli intervalli tipici per i numeri interi su macchine a 64 bit (soprattutto sotto UNIX).

**Tabella 7.3** I tipi interi su una macchina a 64 bit

| Tipo               | Valore più piccolo         | Valore più grande          |
|--------------------|----------------------------|----------------------------|
| short int          | -32,768                    | 32,767                     |
| unsigned short int | 0                          | 65,535                     |
| int                | -2,147,483,648             | 2,147,483,647              |
| unsigned int       | 0                          | 4,294,697,295              |
| long int           | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,807  |
| unsigned long int  | 0                          | 18,446,744,073,709,551,615 |

È bene sottolineare ancora una volta che gli intervalli indicati nelle Tabelle 7.1, 7.2 e 7.3 non sono stabiliti dallo standard C e possono variare da un compilatore a un altro. Un modo per determinare l'intervalle coperto dai vari tipi interi su una particolare implementazione è quello di controllare l'header `<limits.h>` [header <limits.h> 23.2]. Questo header, che fa parte della libreria standard, definisce delle macro per la rappresentazione del più piccolo e del più grande valore dei diversi tipi interi.

(C99)

## Tipi interi nel C99

Il C99 fornisce due tipi interi aggiuntivi: `long long int` e `unsigned long long`. Questi tipi sono stati aggiunti per la crescente necessità di numeri interi molto grandi e per la capacità dei nuovi processori di supportare l'aritmetica a 64 bit. Entrambe le tipi `long long` devono contenere almeno 64 bit e quindi l'intervallo dei valori per `long long int` va tipicamente da  $-2^{63}$  (-9.223.372.036.854.775.808) a  $2^{63}-1$  (9.223.372.036.854.775.807). L'intervallo per una variabile `unsigned long long int`, invece, è tipicamente compreso tra 0 e  $2^{64}-1$  (18.446.744.073.709.551.615).

I tipi `short int`, `int`, `long int` e `long long int` (assieme al tipo `signed char` [[tipo signed char > 7.3](#)]) vengono chiamati dallo standard C99 come **standard signed integer types**. I tipi `unsigned short int`, `unsigned int`, `unsigned long int` e `unsigned long long int` (assieme al tipo `unsigned char` [[tipo unsigned char > 7.3](#)]) e al tipo `_Bool` [[tipo \\_Bool > 5.2](#)]) vengono chiamati **standard unsigned integer types**.

In aggiunta alle tipologie standard, il C99 permette la definizione da parte dell'implementazione dei cosiddetti **extended integer types**, che possono essere sia `signed` che `unsigned`. Un compilatore può fornire per esempio dei tipi `signed` e `unsigned` di 128 bit.

## Costanti intere

Poniamo ora la nostra attenzione sulle **costanti** (numeri che appaiono nel testo di un programma, non numeri che vengono letti, scritti o calcolati). Il C permette la scrittura di costanti intere in formato decimale (base 10), ottale (base 8) o esadecimale (base 16).

## Numeri ottali ed esadecimali

Un numero ottale viene scritto usando solamente le cifre che vanno da 0 a 7. In un numero ottale, ogni posizione rappresenta una potenza di 8 (proprio come in decimale ogni posizione rappresenta una potenza di 10). Di conseguenza il numero ottale 237 rappresenta il numero decimale  $2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0 = 128 + 24 + 7 = 159$ .

Un numero esadecimale è scritto usando le cifre che vanno da 0 a 9 e le lettere dalla A alla F, le quali valgono rispettivamente 10 e 15. In un numero esadecimale, ogni posizione rappresenta una potenza di 16. Il numero esadecimale 1AF equivale al decimale  $1 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 256 + 160 + 15 = 431$ .

- Le costanti **decimali** contengono cifre comprese tra 0 e 9 e non devono iniziare per 0:  
15 255 32767
- Le costanti **ottali** contengono cifre comprese tra 0 e 7 e *devono* iniziare per 0:  
017 0377 077777
- Le costanti **esadecimali** contengono cifre comprese tra 0 e 9 e lettere dalla a alla f, inoltre devono necessariamente iniziare per 0x:  
0xf 0xffff 0x7fff

Le lettere presenti nelle costanti esadecimale possono essere sia maiuscole che minuscole:

```
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
```

Tenete presente che il sistema ottale e quello esadecimale non sono altro che un modo alternativo di scrivere i numeri, non hanno alcun effetto sul modo in cui i numeri vengono memorizzati (gli interi vengono sempre memorizzati in binario, indipendentemente dalla notazione usata per esprimere). Possiamo passare da una notazione all'altra in ogni momento e persino mescolare le notazioni:  $10 + 015 + 0x20$  vale 55 (in decimale). Le notazioni ottale ed esadecimale sono per lo più convenienti nella scrittura di programmi a basso livello, non le useremo molto almeno fino al Capitolo 20.

Solitamente una costante *decimale* è di tipo int. Tuttavia se il valore della costante è troppo grande per essere memorizzato come un int, questa diventa di tipo long int. Nel raro caso in cui una costante fosse troppo grande per venir memorizzata come un long int, il compilatore tenterebbe il tipo unsigned long int come ultima risorsa. Le regole per determinare il tipo delle costanti ottali ed esadecimale sono leggermente diverse: il compilatore passa attraverso tutti i tipi int, unsigned int, long int e unsigned long int fino a che non ne trova uno in grado di rappresentare la costante in esame.

Per forzare il compilatore a considerare una costante come un long int, è sufficiente far seguire questa dalla lettera L (o l):

```
15L 0377L 0x7fffL
```

Per indicare invece che una costante è di tipo unsigned, si deve usare una lettera U (o u):

```
15U 0377U 0x7fffU
```

Le lettere U ed L possono essere usate congiuntamente per indicare che una costante è sia di tipo long che di tipo unsigned: 0xffffffffFUL (l'ordine di L e U non ha importanza e non ne ha nemmeno il caso).

## C99 Costanti intere nel C99

Nel C99 le costanti che terminano con LL o ll (le due lettere devono essere entrambe maiuscole o minuscole) sono di tipo long long int. Aggiungere una lettera U (o u) prima o dopo l'indicazione LL o ll, fa sì che la costante sia di tipo unsigned long long int.

Le regole del C99 per determinare il tipo di una costante sono leggermente diverse rispetto a quelle del C89. Il tipo di una costante decimale sprovvista di suffisso (U, u, L, l, LL o ll) è il più piccolo tra i tipi int, long int o long long int che è in grado di rappresentarla. Per le costanti ottali ed esadecimale però, la lista dei possibili tipi è nell'ordine: int, unsigned int, long int, unsigned long int, long long int e unsigned long long int. Un qualsiasi suffisso posto alla fine di una costante modifica la lista dei tipi ammissibili. Per esempio, una costante che termina con U (o u) deve assumere uno tra i tipi unsigned int, unsigned long int e unsigned long long int. Una costante decimale che termina con una L (o una l) deve essere di tipo long int o long long int.

## Integer overflow

Quando vengono effettuate operazioni aritmetiche sui numeri interi, c'è la possibilità che il risultato sia troppo grande per essere rappresentato. Per esempio quando un'operazione aritmetica viene eseguita su due valori di tipo `int`, il risultato deve anche esso essere rappresentabile come un `int`. Nel caso questo non fosse possibile (perché richiede un numero maggiore di bit), diciamo che si è verificato un **overflow**.

Il comportamento a seguito di un overflow tra interi dipende dal fatto che gli operandi siano con o senza segno. Quando, durante delle operazioni tra interi *con segno*, si verifica un overflow, il comportamento del programma non è definito. Nella Sezione 4.4 abbiamo dato che le conseguenze del comportamento indefinito possono variare. La cosa più probabile è che il risultato dell'operazione sia semplicemente errato, tuttavia il programma potrebbe andare in crash o esibire un comportamento inatteso.

Quando durante delle operazioni su numeri *senza segno* si verifica un overflow, sebbene il comportamento sia definito, otteniamo il risultato in modulo  $2^n$ , dove  $n$  è il numero di bit usati per memorizzare il risultato. Per esempio, se al numero `unsigned` su 16 bit 65,535 sommiamo 1, abbiamo la garanzia che il risultato sia pari a 0.

## Leggere e scrivere interi

Supponete che un programma non stia funzionando a causa di un overflow su una variabile di tipo `int`. Il nostro primo pensiero sarebbe quello di cambiare il tipo della variabile da `int` a `long int`. Questo, tuttavia, non è sufficiente. Dobbiamo infatti controllare gli effetti che questa modifica avrà sul resto del programma. In particolare dobbiamo controllare se la variabile viene utilizzata in chiamate alle funzioni `printf` e `scanf`. Se così fosse, allora dovremmo cambiare la stringa di formato dato che la specifica di conversione `%d` funziona solo con il tipo `int`.

Leggere e scrivere interi `unsigned`, `short` e `long` richiede diverse nuove specifiche di conversione.

**D&R**

- Quando leggiamo o scriviamo un intero `unsigned` dobbiamo usare le lettere `u`, o oppure `x` al posto della specifica di conversione `d`. Se è presente la specifica `u`, il numero viene letto (o scritto) in notazione decimale. La specifica `o` indica la notazione ottale, mentre la specifica `x` indica la notazione esadecimale.

```
unsigned int u;

scanf("%u", &u); /* legge u in base 10 */
printf("%u", u); /* scrive u in base 10 */
scanf("%o", &u); /* legge u in base 8 */
printf("%o", u); /* scrive u in base 8 */
scanf("%x", &u); /* legge u in base 16 */
printf("%x", u); /* scrive u in base 16 */
```

- Quando viene letto o scritto un intero `short`, si deve inserire una lettera `h` come prefisso alle lettere `d`, `o`, `u` o `x`:

```
short s;

scanf("%hd", &s);
printf("%hd", s);
```

- Quando viene letto o scritto un intero *long*, si deve inserire una lettera l come prefisso alle lettere d, o, u o x:

```
long l;
scanf("%ld", &l);
printf("%ld", l);
```

C99

- Quando viene letto o scritto un intero *long long* (solo per il C99), si deve inserire la combinazione di lettere ll come prefisso alle specifiche d, o, u o x:

```
long long ll;
scanf("%lld", &ll);
printf("%lld", ll);
```

## PROGRAMMA

**Sommare una serie di numeri (rivisitato)**

Nella Sezione 6.1 abbiamo scritto un programma che è in grado di sommare una serie di numeri interi immessi dall'utente. Un problema di questo programma è che la somma (o uno dei numeri di input) può eccedere il limite del massimo numero rappresentabile con una variabile int. Ecco cosa potrebbe succedere se il programma venisse eseguito su una macchina i cui interi sono lunghi 16 bit:

```
This program sums a series of integers.
Enter integers (0 to terminate): 10000 20000 30000 0
The sum is: -5536
```

Il risultato della somma era 60,000 che è un numero non rappresentabile con una variabile int e per questo motivo si è verificato un overflow. Quando l'overflow si verifica durante un'operazione con numeri con segno, l'esito non è definito. In questo caso otteniamo un numero che è apparentemente privo di senso. Per migliorare il programma modifichiamolo usando variabili di tipo long.

```
sum2.c /* Somma una serie di numeri(usando variabili long) */
#include <stdio.h>
int main(void)
{
 long n, sum = 0;
 printf("This program sums a series of integers.\n");
 printf("Enter integers (0 to terminate): ");
 scanf("%ld", &n);
 while (n != 0) {
 sum += n;
 scanf("%ld", &n);
 }
 printf("The sum is: %ld\n", sum);
 return 0;
}
```

La modifica è piuttosto semplice: abbiamo dichiarato `n` e `sum` come variabili long invece di int, successivamente abbiamo cambiato le specifiche di conversione per la scanf e la printf usando `%ld` al posto di `%d`.

## 7.2 Tipi floating point

Gli interi non sono appropriati per tutti i tipi di applicazioni. A volte può essere necessario usare variabili in grado di immagazzinare numeri con delle cifre dopo la virgola, oppure numeri che sono eccezionalmente grandi o piccoli. Tali numeri vengono memorizzati nel formato a virgola mobile (chiamato così perché il separatore decimale è "flottante"). Il C fornisce tre **tipi floating point**, corrispondenti a differenti formati:

|                          |                                      |
|--------------------------|--------------------------------------|
| <code>float</code>       | floating point a singola precisione  |
| <code>double</code>      | floating point a doppia precisione   |
| <code>long double</code> | floating point con precisione estesa |

il tipo `float` è appropriato per i casi in cui la precisione non è un fattore critico (per esempio quando si calcolano temperature con una sola cifra decimale). Il tipo `double` fornisce una precisione maggiore (sufficiente per la maggior parte dei programmi). Il tipo `long double`, che fornisce la precisione più grande, viene usato raramente nella pratica.

Lo standard C non specifica quale debba essere la precisione dei tipi `float`, `double` e `long double` dato che computer diversi potrebbero memorizzare i numeri a virgola mobile in modi differenti. I computer più moderni seguono le specifiche degli standard IEEE Standard 754 (conosciuto come IEC 60559), per questo motivo useremo questa specifica come esempio.

### Lo standard floating point dell'IEEE

Lo standard IEEE 754 sviluppato dall'*Institute of Electrical and Electronics Engineers*, prevede due formati principali per i numeri a virgola mobile: singola precisione (32 bit) e doppia precisione (64 bit). I numeri vengono memorizzati seguendo una notazione scientifica, dove ogni numero è costituito da tre parti: il **segno**, l'**esponente**, e la **mantissa**. Il numero di bit riservato per l'esponente determina quanto grandi (e quanto piccoli) possono essere i numeri. Nei numeri a precisione singola l'esponente è lungo 8 bit mentre la mantissa occupa 23 bit. Ne risulta che i numeri a singola precisione hanno un valore massimo corrispondente all'incirca a  $3.40 \times 10^{38}$ , con una precisione di circa 6 cifre decimali.

Lo standard IEEE descrive anche altri due formati: la precisione singola estesa e la precisione doppia estesa. Lo standard non specifica il numero di bit di questi formati ma impone che il tipo a singola precisione estesa occupi almeno 43 bit e che il tipo a doppia precisione estesa ne occupi almeno 79. Per maggiori informazioni sullo standard IEEE e sull'aritmetica floating point in generale leggete "What every computer scientist should know about floating-point arithmetic" di David Goldberg (ACM Computing Surveys, vol 23, no. 1 (marzo 1991): 5-48).

La Tabella 7.4 illustra le caratteristiche dei tipi a virgola mobile implementati in accordo allo standard IEEE (la tabella mostra i numeri positivi più piccoli *normalizzati*. I

numeri non normalizzati possono essere più piccoli [numeri non normalizzati > 23.4].) Il tipo long double non è incluso nella tabella dato che la sua lunghezza varia da una macchina all'altra (80 bit e 128 bit sono le dimensioni più comuni per questo tipo).

**Tabella 7.4** Caratteristiche dei tipi floating point (Standard IEEE)

| Tipo   | Valore Positivo Più Piccolo | Valore Più Grande      | Precisione |
|--------|-----------------------------|------------------------|------------|
| float  | $1.17549 \times 10^{-38}$   | $3.40282 \times 1038$  | 6 digits   |
| double | $2.22507 \times 10^{-308}$  | $1.79769 \times 10308$ | 15 digits  |

La Tabella 7.4 non è valida per i computer che non seguono lo standard IEEE. Di fatto su alcune macchine il tipo float può avere lo stesso insieme di valori di un double, o un double può avere lo stesso insieme di valori di un long double. Le macro che definiscono le caratteristiche dei numeri floating point possono essere trovate nell'header <float.h> [header <float.h> > 23.1].

Nel C99 i tipi a virgola mobile sono suddivisi in due categorie. I tipi float, double e long double ricadono dentro la categoria chiamata dei **floating point reali**. I tipi a virgola mobile, includono anche i **tipi floating point complessi** (float\_Complex, double\_Complex e long double\_Complex) che sono una novità dello standard C99 [tipi floating point complessi > 27.3].

## Costanti floating point

Le costanti floating point possono essere scritte in molti modi. Le seguenti costanti, per esempio, rappresentano tutte delle modalità ammesse per scrivere il numero 57.0:

57.0 57. 57.0e0 57E0 5.7e1 5.7e+1 .57e2 570.e-1

Una costante floating point deve contenere il separatore decimale e/o un esponente. L'esponente indica la potenza di 10 alla quale deve essere moltiplicato il numero. Se è presente un esponente, questo deve essere preceduto dalla lettera E (o e). Optionalmente può essere usato un segno dopo la lettera E (o e).

Per default le costanti floating point vengono memorizzate come numeri a precisione doppia. In altre parole, quando un compilatore C trova la costante 57.0 all'interno di un programma, fa in modo che il numero venga immagazzinato in memoria nello stesso formato di una variabile double. Generalmente questo non causa problemi dato che i valori double vengono convertiti automaticamente nel tipo float se necessario.

Occasionalmente potrebbe essere necessario forzare il compilatore a usare per una costante il formato float o quello long double. Per indicare che si desidera la precisione singola si deve mettere un lettera F (o f) alla fine della costante (per esempio 57.0F). Per indicare invece che la costante deve essere memorizzata con il formato long double, si deve mettere la lettera L (o l) alla fine della costante (57.0L).

Il C99 prevede la possibilità di scrivere costanti a virgola mobile nel formato esadecimale. Queste costanti andranno espresse facendole precedere da 0x o 0X (esattamente come avviene per le costanti esadecimali intere). Questa funzionalità dello standard tuttavia viene utilizzata molto di rado.



## Leggere e scrivere numeri a virgola mobile

Come abbiamo dato precedentemente, le specifiche di conversione %e, %f e %g vengono utilizzate per leggere e scrivere i numeri floating point a singola precisione. Valori di tipo double o long double richiedono delle conversioni leggermente diverse.

- Per leggere un valore di tipo double, si deve mettere una lettera l come prefisso alle lettere e, f o g:

```
double d;
```

```
scanf("%lf", &d);
```

D&R

*Nota:* usate la l solo nelle stringhe di formato delle scanf, non in quelle della printf. Nelle stringhe di formato per le printf le specifiche di conversione e, f e g possono essere utilizzate sia per valori float che per valori double. (Il C99 ammette l'uso di %le, %lf e %lg nelle chiamate alle printf, sebbene la l non abbia alcun effetto.)

- Per leggere o scrivere un valore di tipo long double, si deve mettere una lettera L come prefisso alle lettere e, f o g:

```
long double ld;
```

```
scanf("%Lf", &ld);
printf("%Lf", ld);
```

C99

## 7.3 Tipi per i caratteri

D&R

L'unico tipo di base che è rimasto è il char, il tipo per i caratteri. I valori del tipo char possono variare da computer a computer a causa del fatto che le varie macchine possono basarsi su un diverso set di caratteri.

### Set di caratteri

Attualmente il set di caratteri più diffuso è quello ASCII (*American Standard Code for Information Interchange*) [set dei caratteri ASCII > Appendice D], un codice a 7 bit capace di rappresentare 128 caratteri diversi. In ASCII le cifre da 0 a 9 vengono rappresentate da codici che vanno da 0110001 a 0111001, mentre le lettere maiuscole dalla A alla Z sono rappresentate dal codice 1000001 fino al codice 1011010. Il codice ASCII spesso viene esteso a un codice a 256 caratteri chiamato Latin-1 che prevede i caratteri necessari per le lingue dell'Europa Occidentale e molte lingue dell'Africa.

A una variabile di tipo char può essere assegnato un qualsiasi carattere:

```
char ch;
ch = 'a'; /* a minuscola */
ch = 'A'; /* A maiuscola */
ch = '0'; /* zero */
ch = ' '; /* spazio */
```

Osservate che le costanti di tipo carattere sono racchiuse da apici singoli e non doppi.

## Operazioni sui caratteri

Lavorare con i caratteri è piuttosto semplice grazie al fatto che il C tratta i caratteri come dei piccoli interi. Dopo tutto i caratteri sono codificati in binario e non ci vuole molta immaginazione per vedere questi codici binari come numeri interi. Nello standard ASCII, per esempio, l'intervallo dei codici per i caratteri va da 00000000 fino a 11111111, e questi possono essere pensati come gli interi da 0 a 127. Il carattere 'a' ha il valore 97, 'A' ha il valore 65, '0' ha il valore 48 e ' ' ha il valore 32. La connessione tra caratteri e numeri interi è così forte nel C che attualmente le costanti carattere sono di tipo int invece che char (un fatto interessante, ma del quale nella maggior parte dei casi non ci preoccuperemo affatto).

Quando un carattere compare all'interno di un calcolo, il C utilizza semplicemente il suo valore intero. Considerate gli esempi seguenti che presumono l'uso del set di caratteri ASCII:

```
char ch;
int i;
i = 'a'; /* adesso i è uguale a 97 */
ch = 65; /* adesso ch è uguale a 'A' */
ch = ch + 1; /* adesso ch è uguale a 'B' */
ch++; /* adesso ch è uguale a 'C' */
```

I caratteri possono essere confrontati esattamente come accade per gli interi. La seguente istruzione if controlla se il carattere ch contiene una lettera minuscola, in tal caso converte ch in una lettera maiuscola.

```
if ('a' <= ch && ch <= 'z')
 ch = ch - 'a' + 'A';
```

I confronti come 'a' <= ch vengono fatti utilizzando i valori interi dei caratteri coinvolti. Questi valori dipendono dal set di caratteri in uso, di conseguenza i programmi che usano <, <=, > e >= per il confronto dei caratteri non sono portabili.

Il fatto che i caratteri abbiano le stesse proprietà dei numeri porta ad alcuni vantaggi. Per esempio possiamo scrivere facilmente un ciclo for la cui variabile di controllo salta attraverso tutte le lettere maiuscole:

```
for (ch = 'A'; ch <= 'Z'; ch++) ...
```

D'altro canto trattare i caratteri come numeri può portare a diversi errori di programmazione che non verranno individuati dal compilatore e ci permette di scrivere 'a' \* 'b' / 'c'. Questo comportamento può rappresentare un ostacolo per la portabilità dato che i nostri programmi potrebbero essere basati su assunzioni riguardanti il set di caratteri presente (il nostro ciclo for, per esempio, assume che i codici delle lettere dalla 'A' alla 'Z' siano consecutivi).

## Caratteri signed e unsigned

Considerato che il C permette di usare i caratteri come numeri interi non deve sorprendervi il fatto che il tipo char (come gli altri tipi interi) sia presente sia nella

versione signed che in quella unsigned. Tipicamente i caratteri di tipo signed hanno valori compresi tra -128 e 127, mentre i caratteri unsigned hanno valori tra 0 e 255.

Lo standard C non specifica se il tipo char ordinario debba essere di tipo signed o unsigned, alcuni compilatori lo trattano in un modo, altri nell'altro (alcuni persino permettono al programmatore di scegliere, attraverso le opzioni del compilatore, se il tipo char debba essere con o senza segno).

### D&R

La maggior parte delle volte non ci cureremo del fatto che il tipo char sia con o senza segno. In certi casi però saremo costretti a farlo, specialmente se stiamo utilizzando una variabile char per memorizzare dei piccoli interi. Per questa ragione il linguaggio permette di utilizzare le parole signed e unsigned per modificare il tipo char:

```
signed char sch;
unsigned char uch;
```

### PORTABILITÀ

*Non fate supposizioni riguardo al fatto che il carattere char sia per default con o senza segno. Se avesse importanza utilizzate le diciture signed char o unsigned char al posto del semplice char.*

Alla luce della stretta relazione esistente tra i caratteri e gli interi, il C89 usa il termine tipi integrali (*integral types*) per riferirsi a entrambi. Anche i tipi enumerati fanno parte dei tipi integrali [tipi enumerati > 16.5].

### C99

Il C99 non usa il termine "integral types", ma al suo posto invece espande il concetto di tipi interi (*integer types*) per includere i caratteri e i tipi enumerati. Il tipo \_Bool è considerato come un integer type di tipo unsigned [tipo \_Bool > 5.2].

## Tipi aritmetici

I tipi interi e quelli a virgola mobile sono conosciuti collettivamente come tipi aritmetici (*arithmetic types*). Ecco un sommario dei tipi aritmetici del C89 diviso in categorie e sottocategorie:

- Integral types
  - char
  - Tipi interi con segno (signed char, short int, int, long int)
  - Tipi interi senza segno (unsigned char, unsigned short int, unsigned int, unsigned long int)
  - Tipi enumerati
  - Tipi floating point (float, double, long double)

### C99

Il C99 possiede una gerarchia più complicata per i suoi tipi aritmetici:

- Tipi interi
  - char
  - Tipi interi con segno sia standard che estesi (signed char, short int, int, long int, long long int)
  - Tipi interi senza segno sia standard che estesi (unsigned char, unsigned short int, unsigned int, unsigned long int, unsigned long long int, \_Bool)
  - Tipi enumerati

- Tipi floating point
  - Tipi floating point reali (`float`, `double`, `long double`)
  - Tipi complessi (`float _Complex`, `double _Complex`, `long double _Complex`)

## Sequenze di escape

Così come abbiamo dato negli esempi precedenti, una costante carattere di solito è costituita da un unico carattere racchiuso tra apici singoli. Tuttavia alcuni caratteri speciali (tra cui il carattere new-line) non possono essere scritti in questo modo, perché sono invisibili (non stampabili) o perché non possono essere immessi dalla tastiera. Per fare in modo che i programmi possano utilizzare tutti i tipi di caratteri appartenenti al set installato, il C fornisce una notazione speciale: le **sequenze di escape**.

Ci sono due tipi di sequenze di escape: i **caratteri di escape** e gli **escape numerici**. Abbiamo dato un elenco parziale di escape carattere nella Sezione 3.1. La Tabella 7.5 fornisce il set completo.

**Tabella 7.5** Caratteri di escape

| Nome             | Sequenza di Escape |
|------------------|--------------------|
| Alert (bell)     | \a                 |
| Backspace        | \b                 |
| Form feed        | \f                 |
| New-line         | \n                 |
| Carriage return  | \r                 |
| Tab orizzontale  | \t                 |
| Tab verticale    | \v                 |
| Backslash        | \\\                |
| Punto di domanda | \?                 |
| Apice singolo    | \'                 |
| Apice doppio     | \"                 |

Gli escape \a, \b, \f, \r, \t, e \v rappresentano dei caratteri di controllo ASCII comuni. Il carattere di escape \n rappresenta il carattere ASCII new-line. L'escape \\ permette a una costante-carattere o a una stringa di contenere il carattere \. L'escape \' permette a una costante carattere di contenere il carattere ', mentre l'escape \" permette alle stringhe di contenere il carattere ". Il carattere di escape \? viene usato raramente.

I caratteri di escape sono comodi, tuttavia hanno un problema: non includono tutti i caratteri ASCII non stampabili ma solo i più comuni. I caratteri di escape non permettono nemmeno la rappresentazione dei caratteri che vanno oltre i 128 caratteri di base del codice ASCII. Gli escape numerici, che permettono di rappresentare *qualsiasi* carattere, costituiscono la soluzione a questo tipo di problemi.

Per scrivere un escape numerico per un particolare carattere dobbiamo per prima cosa guardare il suo valore ottale ed esadecimale in una tavola come quella presente

nell'Appendice D. Per esempio, il carattere esc del codice ASCII (valore decimale 27) ha valore 33 in ottale e 1B in esadecimale. Entrambi questi codici possono essere usati per scrivere una sequenza di escape:

- Una **sequenza di escape ottale** consiste del carattere \ seguito da un numero ottale con al più tre cifre (questo numero deve essere rappresentabile come un unsigned char e quindi di solito il suo massimo valore in ottale è 377). Per esempio i caratteri di escape possono essere scritti come \33 o come \033. I numeri ottali delle sequenze di escape (a differenza delle costanti ottali) non devono iniziare per 0.
- Una **sequenza di escape esadecimale** consiste di un numero esadecimale preceduto dal prefisso \x. Sebbene il C non ponga limiti rispetto alla quantità di cifre esadecimali che il numero può avere, questo deve essere rappresentabile come un unsigned char (e quindi non può eccedere oltre FF nel caso in cui i caratteri fossero lunghi otto bit). Utilizzando questa notazione, il carattere escape viene scritto come \x1b oppure come \x1B. La x deve essere minuscola, mentre le cifre esadecimali (come b) possono essere sia maiuscole che minuscole.

Quando vengono usate come costante carattere, le sequenze di escape devono essere rinchuse tra singoli apici. Per esempio, una costante rappresentante il carattere esc dovrebbe essere scritta come '\33' (o '\x1b'). Le sequenze di escape tendono a diventare un po' criptiche, per questo è buona pratica denominarle usando la direttiva #define:

```
#define ESC '\33' /* carattere ESC ASCII */
```

Nella Sezione 3.1 abbiamo dato che le sequenze di escape possono essere incorporate anche all'interno delle stringhe.

C99

Le sequenze di escape non sono solo una notazione speciale per rappresentare i caratteri. Le **sequenze trigrafiche (trigraph sequences)** [sequenze trigrafiche > 25.3] forniscono un modo per rappresentare i caratteri #, [, \, ], ^, {, |, } e ~ che potrebbero non essere disponibili sulle tastiere di alcune nazionalità. Il C99 aggiunge inoltre dei nomi universali per i caratteri che assomigliano alle sequenze di escape. A differenza di queste ultime però, i nomi universali per i caratteri (*universal character names*) [universal character names > 25.4] sono ammessi anche all'interno degli identificatori.

## Funzioni per la manipolazione dei caratteri

Nella sezione precedente abbiamo dato come scrivere un'istruzione if per convertire una lettera minuscola in una maiuscola:

```
if ('a' <= ch && ch <= 'z')
 ch = ch - 'a' + 'A';
```

Questo però non è il metodo migliore per farlo. Un modo più veloce (e più portatile) per convertire il case di un carattere è quello di chiamare la funzione toupper appartenente alla libreria del C.

```
ch = toupper(ch); /* converte ch in una lettera maiuscola */
```

Quando viene chiamata, la funzione toupper controlla se il suo argomento (ch in questo caso) è una lettera minuscola. Se è così, la funzione toupper restituisce la lettera maiuscola corrispondente, altrimenti viene restituito il valore del suo argomento. Nel nostro esempio abbiamo utilizzato l'operatore di assegnamento per memorizzare all'interno della variabile ch il valore di restituito dalla funzione toupper. In realtà avremmo potuto facilmente eseguire altre operazioni come memorizzare il valore di ritorno in un'altra variabile oppure analizzarlo all'interno di un if:

```
if (toupper(ch) == 'A') ...
```

I programmi che richiamano la funzione toupper hanno bisogno della seguente direttiva:

```
#include <ctype.h>
```

La toupper non è l'unica funzione utile per la manipolazione dei caratteri presente nella libreria del C. La Sezione 23.5 le descrive tutte e fornisce degli esempi sul loro utilizzo.

## Leggere e scrivere caratteri usando le funzioni scanf e printf

La specifica di conversione %c permette alla scanf e alla printf di leggere e scrivere singoli caratteri:

```
char ch;
scanf("%c", &ch); /* legge un singolo carattere */
printf("%c", ch); /* scrive un singolo carattere */
```

La funzione scanf non salta i caratteri di spazio bianco prima della lettura di un carattere. Se il successivo carattere non letto è uno spazio, allora la variabile ch dell'esempio precedente conterrà uno spazio dopo il ritorno della funzione scanf. Per forzare la scanf a saltare gli spazi prima della lettura di un carattere si deve mettere uno spazio all'interno della stringa di formato esattamente prima al %c:

```
scanf(" %c", &ch); /* salta gli spazi e poi legge ch */
```

Vi ricorderete dalla Sezione 3.2 che uno spazio in una stringa di formato di una scanf significa "salta zero o più spazi bianchi".

Dato che la scanf di norma non salta gli spazi, è facile trovare la fine di una riga di input: è sufficiente controllare se il carattere appena letto è un carattere di newline. Per esempio, il ciclo seguente leggerà e ignorerà tutti i caratteri rimanenti nella corrente riga di input:

```
do {
 scanf("%c", &ch);
} while (ch != '\n');
```

La prossima volta che la scanf verrà chiamata, leggerà il primo carattere della riga di input successiva.

## Leggere e scrivere caratteri usando le funzioni getchar e putchar

**D&R** Il C fornisce altri modi per leggere e scrivere un singolo carattere. In particolare possiamo usare le funzioni `getchar` e `putchar` invece di chiamare le funzione `scanf` e `printf`. La funzione `putchar` scrive un singolo carattere:

```
putchar(ch);
```

Ogni volta che la funzione `getchar` viene chiamata, questa legge un carattere che poi restituisce. Per salvare questo carattere in una variabile dobbiamo fare un assegnazione:

```
ch = getchar(); /* legge un carattere e lo salva in ch */
```

In effetti `getchar` restituisce un valore di tipo `int` invece che un valore `char` (la ragione verrà discussa nei capitoli seguenti). Questo è il motivo per cui non è affatto raro trovare variabili `int` utilizzate per memorizzare caratteri letti con la funzione `getchar`. Esattamente come la `scanf`, anche la funzione `getchar` non salta gli spazi bianchi mentre legge dall'input.

Usare `getchar` e `putchar` (invece che `scanf` e `printf`) permette di risparmiare tempo durante l'esecuzione del programma. Le due funzioni sono veloci per due ragioni. La prima è che sono molto più semplici rispetto alla `scanf` e alla `printf` che sono state progettate per leggere e scrivere molti tipi di dati, secondo una varietà di formati diversi. La seconda è che di solito la `getchar` e la `putchar` vengono implementate come delle macro [macro > 14.3] per una maggiore velocità.

La `getchar` inoltre ha un altro vantaggio rispetto alla `scanf`: dato che restituisce il carattere letto, la `getchar` si presta a diversi idiom del C, inclusi i cicli per la ricerca di un carattere o di tutte le sue occorrenze. Considerate il ciclo `scanf` che abbiamo usato per saltare la parte rimanente di una riga di input:

```
do {
 scanf("%c", &ch);
} while (ch != '\n');
```

Riscrivendolo usando la `getchar` otteniamo il seguente codice:

```
do {
 ch = getchar();
} while (ch != '\n');
```

Spostare la chiamata alla `getchar` all'interno dell'espressione di controllo ci permette di condensare ulteriormente il ciclo:

```
while ((ch = getchar()) != '\n')
;
```

Questo ciclo legge un carattere, lo salva nella variabile `ch` e poi controlla se è diverso dal carattere new-line. Se il test ha esito positivo viene eseguito il corpo del ciclo (che è vuoto). Successivamente il controllo del ciclo viene rieseguito causando

la lettura di un nuovo carattere. Agli effetti pratici non abbiamo nemmeno bisogno della variabile ch, infatti possiamo semplicemente confrontare il valore restituito dalla getchar con il carattere new-line:

```
while (getchar() != '\n') /* salta il resto della riga */
;
```

Il ciclo che ne risulta è un idioma del C molto conosciuto, un po' ma che è bene conoscere.

La funzione getchar è utile per i cicli che saltano i caratteri ma lo è anche per i cicli che vanno alla ricerca di particolari caratteri. Considerate l'istruzione seguente che usa la getchar per saltare un numero indefinito di caratteri di spazio:

```
while ((ch = getchar()) == ' ') /* salta gli spazi */
;
```

Quando il ciclo ha termine, la variabile ch contiene il primo carattere non bianco che viene incontrato dalla getchar.



Fate attenzione se mischiate la getchar e la scanf all'interno dello stesso programma. La scanf ha la tendenza a lasciarsi alle spalle i caratteri che prende ma non legge, inclusi i caratteri new-line. Considerate cosa succederebbe se prima cercassimo di leggere un numero e poi un carattere:

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

La chiamata alla scanf si lascia alle spalle alcuni caratteri che non sono stati consumati durante la lettura di i, incluso (ma non solo) il carattere new-line. La getchar caricherà il primo carattere lasciato indietro e questo non era certo quello che avevamo in mente.

## PROGRAMMA

### Determinare la lunghezza di un messaggio

Per illustrare come vengono letti i caratteri, scriviamo un programma che calcola la lunghezza di un messaggio. Dopo che l'utente ha immesso il messaggio, il programma visualizza la sua lunghezza:

```
Enter a message: Brevity is the soul of wit.
Your message was 27 character(s) long.
```

La lunghezza include anche gli spazi e i caratteri di interpunkzione, ma non il carattere new-line presente alla fine del messaggio.

Abbiamo bisogno di un ciclo il cui corpo legga un carattere e contestualmente incrementi un contatore. Il ciclo dovrà terminare non appena viene incontrato il carattere new-line. Possiamo usare sia la scanf che la getchar per leggere i caratteri, ma molti programmati C sceglierrebbero la getchar. Usando un opportuno ciclo while possiamo ottenere il seguente programma.

```
length.c /* Determina la lunghezza di un messaggio */

#include <stdio.h>

int main(void)
{
 char ch;
 int len = 0;

 printf("Enter a message: ");
 ch = getchar();
 while (ch != '\n') {
 len++;
 ch = getchar();
 }
 printf("Your message was %d character(s) long.\n", len);

 return 0;
}
```

Ricordando la nostra discussione sugli idiom che coinvolgono i cicli while e la getchar capiamo che il programma può essere abbreviato:

```
length2.c /* Determina la lunghezza di un messaggio */

#include <stdio.h>

int main(void)
{
 int len = 0;

 printf("Enter a message: ");
 while (getchar() != '\n')
 len++;
 printf("Your message was %d character(s) long.\n", len);

 return 0;
}
```

## 7.4 Conversione di tipo

I computer tendono a essere più restrittivi del C riguardo l'aritmetica. Un computer per poter eseguire un'operazione aritmetica, deve avere operandi della stessa dimensione (lo stesso numero di bit) e memorizzati allo stesso modo. Un computer può sommare direttamente due interi a 16 bit, ma non un intero a 16 bit con uno a 32 bit e lo stesso vale per un intero a 32 bit con un numero a virgola mobile a 32 bit.

Il C d'altra parte permette ai tipi base di essere mischiati all'interno delle espressioni. Possiamo combinare assieme in una sola espressione interi, numeri a virgola mobile e persino caratteri. Quindi affinché l'hardware possa calcolare l'espressione il compilatore C deve generare delle istruzioni che convertono alcuni operandi in un tipo diverso. Per esempio se sommiamo uno short a 16 bit con un int a 32 bit il compilatore farà in modo che il valore dello short venga convertito a 32 bit. Se

sommiamo un int e un float, allora il compilatore deve convertire il valore int nel formato float. Questa conversione è un po' più complicata a causa del fatto che valori int e float vengono salvati in modi completamente diversi.

Per questo il compilatore applica queste conversioni automaticamente senza l'intervento del programmatore, queste vengono dette **conversioni implicite**. Il C permette anche al programmatore di effettuare delle **conversioni esplicite** usando l'operatore di casting. Prima discuteremo delle conversioni implicite mentre ci occuperemo di quelle esplicite in un secondo momento. Sfortunatamente le regole associate alle conversioni implicite sono complesse. Ciò è dovuto al fatto che il C ha molti tipi aritmetici.

Le conversioni implicite avvengono nelle seguenti situazioni:

- Quando in un'espressione logica o aritmetica gli operandi non sono dello stesso tipo (il C effettua quelle che sono conosciute come normali conversioni aritmetiche o **usual arithmetic conversions**).
- Quando il tipo del lato destro di un assegnazione non combacia con quello del lato sinistro.
- Quando il tipo di un argomento passato a una funzione non combacia con quello del parametro corrispondente.
- Quando il tipo di un'espressione in una return non combacia con il tipo di ritorno della funzione.

Per ora discuteremo dei primi due casi, mentre vedremo gli altri nel Capitolo 9.

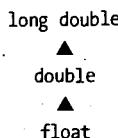
## Le normali conversioni aritmetiche

Le normali conversioni aritmetiche (*usual arithmetic conversions*) vengono applicate agli operandi della maggior parte degli operatori, inclusi quelli aritmetici, quelli relazionali e quelli di uguaglianza. Per esempio, diciamo che *f* è di tipo float mentre *i* è di tipo int. Le normali conversioni aritmetiche vengono applicate agli operandi dell'espressione *f + i* perché questi non sono dello stesso tipo. Chiaramente è più sicuro convertire *i* nel tipo float (facendo corrispondere la variabile al tipo di *f*) piuttosto che convertire *f* nel tipo int (facendola così corrispondere al tipo di *i*). Un intero può essere sempre convertito in un numero a virgola mobile, la cosa peggiore che può capitare è una piccola perdita di precisione. Al contrario, convertire un numero floating point in un int comporterebbe la perdita della parte frazionaria del numero. Peggio ancora, se il numero originale fosse maggiore del più grande numero intero o minore del più piccolo intero, in tal caso il risultato sarebbe completamente privo di significato.

La strategia alla base delle normali conversioni aritmetiche è quella di convertire gli operandi nel tipo "più piccolo" che sia in grado di conciliare con sicurezza entrambi i valori (parlando in modo spicciolo, possiamo dire che un tipo è più piccolo di un altro se richiede meno byte per essere memorizzato). Spesso il tipo degli operandi può essere fatto combaciare convertendo l'operando di tipo più piccolo nel tipo dell'altro operando (questa azione viene detta **promozione**). Tra le promozioni più comuni ci sono le **promozioni integrali** che convertono un carattere o un intero short nel tipo int (o unsigned int in alcuni casi).

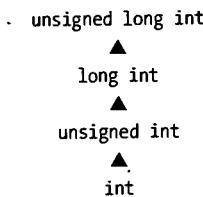
Possiamo suddividere le regole per l'esecuzione delle normali conversioni aritmetiche in due casi.

- **Uno dei due operandi appartiene a uno dei tipi floating point.** Viene promosso l'operando con il tipo più piccolo in accordo con il seguente diagramma:



Questo significa che se uno dei due operandi è di tipo long double, allora l'altro operando viene convertito al tipo long double. Se invece uno dei due operandi è di tipo double, l'altro viene convertito al tipo double. Se uno degli operandi è di tipo float, allora l'altro operando viene convertito al tipo float. Osservate che queste regole riguardano situazioni in cui tipi interi e a virgola mobile sono mischiati: se per esempio uno degli operandi è di tipo long int, mentre l'altro è di tipo double, allora l'operando long int viene convertito in un double.

- **Nessuno dei due operandi appartiene a uno dei tipi floating point.** Per prima cosa viene eseguita una promozione integrale di entrambi gli operandi (garantendo che nessuno dei due sia un carattere o un intero short). Successivamente viene usato lo schema seguente per promuovere l'operando il cui tipo è il più piccolo:



C'è un caso speciale, ma questo accade solamente quando il tipo long int e il tipo unsigned int hanno la stessa dimensione (diciamo 32 bit). In questa circostanza, se uno dei due operandi è di tipo long int e l'altro è di tipo unsigned int, allora entrambi vengono convertiti al tipo unsigned long int.



Quando un operando con segno viene combinato a un operando senza segno, il primo viene convertito in un valore senza segno. La conversione implica la somma o la sottrazione di un multiplo di  $n + 1$ , dove  $n$  è il più grande valore rappresentabile di tipo unsigned. Questa regola può causare oscuri errori di programmazione.

Supponete che una variabile  $i$  di tipo int abbia il valore -10 e che la variabile  $u$  di tipo unsigned abbia valore 10. Confrontando  $i$  e  $u$  con l'operatore  $<$  potremmo aspettarci di ottenere un 1 (true) come risultato. Tuttavia, prima del confronto,  $i$  viene convertita al tipo unsigned int. Dato che un numero negativo non può essere rappresentato come intero unsigned, il valore convertito non sarà -10. Al suo posto viene sommato il valore 4,294,967,296 (assumendo che 4,294,967,295 sia il valore unsigned int più grande), restituendo così un valore convertito pari a 4,294,967,286. Il confronto  $i < u$  produrrà uno 0. Quando un programma tenta di confrontare un numero con segno con uno senza

segno, alcuni compilatori producono un messaggio di warning come *comparison between signed and unsigned*.

È proprio a causa di trappole come questa che è meglio utilizzare il meno possibile gli interi senza segno e soprattutto fare attenzione a non mischiarli mai con gli interi con segno.

L'esempio seguente mostra in azione le normali conversioni aritmetiche:

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

i = i + c; /* c viene convertita al tipo int */
i = i + s; /* s viene convertita al tipo int */
u = u + i; /* i viene convertita al tipo unsigned int */
l = l + u; /* u viene convertita al tipo long int */
ul = ul + l; /* l viene convertita al tipo unsigned long int */
f = f + ul; /* ul viene convertita al tipo float */
d = d + f; /* f viene convertita al tipo double */
ld = ld + d; /* d viene convertita al tipo long double */
```

## Conversioni negli assegnamenti

Le normali conversioni aritmetiche non si applicano alle assegnazioni. In questi casi il C segue la semplice regola di convertire l'espressione presente nel lato destro dell'assegnazione, nel tipo della variabile presente nel lato sinistro. Se il tipo della variabile è "grande" almeno quanto quello dell'espressione, allora il tutto funzionerà senza problemi. Per esempio:

```
char c;
int i;
float f;
double d;
i = c; /* c viene convertita al tipo int */
f = i; /* i viene convertita al tipo float */
d = f; /* f viene convertita al tipo double */
```

Gli altri casi sono problematici. Assegnare un numero a virgola mobile a una variabile intera causa la perdita della parte razionale del numero:

```
int i;
i = 842.97; /* adesso i vale 842 */
i = -842.97; /* adesso i vale -842 */
```

**D&R**

Inoltre, assegnare un valore a una variabile di tipo più piccolo nel caso in cui tale valore fosse al di fuori del range di quest'ultima conduce a un risultato privo di significato (o peggio):

```
c = 10000; /*** SBAGLIATO ***/
i = 1.0e20; /*** SBAGLIATO ***/
f = 1.0el00; /*** SBAGLIATO **/
```

Un'assegnazione "rimpicciolente" può provocare un messaggio di warning da parte del compilatore o da strumenti come lint.

Come abbiamo dato nel Capitolo 2, è una buona pratica aggiungere il suffisso `f` a tutte le costanti a virgola mobile nel caso in cui queste vengano assegnate a variabili `float`:

```
f = 3.14159f;
```

Senza il suffisso, la costante 3.14159 sarebbe di tipo `double`, questo potrebbe essere la causa di messaggio di warning.

**C99**

## Conversioni implicite nel C99

Le regole per le conversioni implicite del C99 in qualche modo sono diverse dalle regole del C89. Questo avviene principalmente a causa dei tipi aggiuntivi (`_Bool` [tipi `_Bool > 5.2`], i tipi `long long`, i tipi interi estesi e i tipi complessi).

Con lo scopo di definire le regole di conversione, il C99 assegna a ogni tipo intero un *rango di conversione intera* (*integer conversion rank*), ovvero un rango di conversione. Ecco i diversi ranghi dal più alto al più basso:

1. `long long int, unsigned long long int`
2. `long int, unsigned long int`
3. `int, unsigned int`
4. `short int, unsigned short int`
5. `char, signed char, unsigned char`
6. `_Bool`

Per semplicità, stiamo ignorando i tipi estesi e quelli enumerati.

In luogo delle promozioni integrali del C89, il C99 ha le "promozioni intere" che coinvolgono la conversione di ogni tipo il cui rango è minore di `int` e `unsigned int` nel tipo `int` (ammesso che tutti i valori di quel tipo possano essere rappresentati come un `int`) oppure nel tipo `unsigned int`.

Come succedeva con il C89, anche nel C99 le regole per le normali conversioni aritmetiche possono essere suddivise in due casi.

- **Uno dei due operandi è di uno dei tipi a virgola mobile.** Se nessuno dei due operandi è di tipo complesso, allora le regole rimangono come quelle già viste (le regole di conversione per i tipi complessi verranno discusse nella Sezione 27.3).
- **Nessuno dei due operandi è di uno dei tipi a virgola mobile.** Per prima cosa viene eseguita la promozione intera su entrambi gli operandi. Se il tipo di

due operandi è uguale allora il processo ha termine. Altrimenti vengono utilizzate le regole che seguono, fermandosi alla prima che può essere applicata.

- Se gli operandi sono entrambi con o senza segno, allora l'operando che ha il rango minore viene convertito al tipo dell'operando con rango maggiore.
- Se un operando senza segno ha rango maggiore o uguale a quello dell'operando con segno, allora quest'ultimo viene convertito al tipo dell'operando senza segno.
- Se il tipo dell'operando con segno può rappresentare tutti i valori del tipo dell'operando senza segno, allora quest'ultimo viene convertito al tipo dell'operando con segno.
- Nei casi rimanenti entrambi gli operandi vengono convertiti al tipo senza segno corrispondente al tipo dell'operando con segno.

A tal proposito, tutti i tipi aritmetici possono essere convertiti nel tipo `_Bool`. Il risultato della conversione è 0 se il valore originale è 0, mentre è 1 negli altri casi.

## Casting

Sebbene le conversioni implicite del C siano convenienti, a volte abbiamo bisogno di un maggior grado di controllo sulla conversione di tipo. Per questa ragione il C fornisce i *cast*. Una espressione di cast ha la forma:

`( nome-del-tipo) espressione`

Il *nome-del-tipo* specifica il tipo nel quale verrà convertita l'espressione.

L'esempio seguente mostra come usare l'espressione di cast per calcolare la parte frazionaria di un valore `float`:

```
float f, frac_part;
frac_part = f - (int) f;
```

L'espressione di cast `(int) f` rappresenta il risultato della conversione del valore di `f` nel tipo `int`. Le normali conversioni aritmetiche del C richiedono quindi che `(int) f` venga convertita nuovamente nel tipo `float` prima di poter effettuare la differenza. La differenza tra `f` e `(int) f` è la parte frazionaria di `f` che è stata persa durante il cast.

Le espressioni di cast ci permettono di documentare le conversioni di tipo che avrebbero avuto luogo in ogni caso:

```
int = (int) f; /* f viene convertita in int */
```

Le espressioni di cast ci permettono inoltre di forzare il compilatore a effettuare le conversioni che vogliamo. Considerare l'esempio seguente:

```
float quotient;
int dividend, divisor;
quotient = dividend / divisor;
```

Per come è stato scritto, il risultato della divisione (un intero) viene convertito in un float prima di essere salvato nella variabile quotient. Tuttavia, per ottenere un risultato più corretto, vorremmo che dividend e divisor venissero convertite in un float *prima* di effettuare la divisione. Un'espressione di cast risolve il problema:

`quotient = (float) dividend / divisor`

divisor non ha bisogno del cast dato che il casting di dividend al tipo float forza il compilatore a convertire anche la variabile divisor allo stesso modo.

Tra l'altro il C tratta l'operatore (*nome-del-tipo*) come un operatore unario. Gli operatori unari hanno precedenza più alta rispetto a quelli binari e quindi il compilatore interpreta

`(float) dividend / divisor`

come

`((float) dividend) / divisor`

Se lo trovate poco chiaro sappiate che ci sono altri modi per ottenere lo stesso effetto:

`quotient = dividend / (float) divisor;`

oppure

`quotient = (float) dividend / (float) divisor;`

I cast a volte sono necessari per evitare gli overflow. Considerate l'esempio seguente:

```
long i;
int j = 1000;
```

```
i = j * j; /* può esserci overflow */
```

A prima vista queste istruzioni sembrano corrette. Il valore di  $j * j$  è 1,000,000 e i di tipo long, per questo dovrebbe essere facile salvare un valore di questa dimensione giusto? Il problema è che quando i due int vengono moltiplicati, il risultato è di tipo int. Ma su certe macchine  $j * j$  è troppo grande per essere rappresentato da un int: questo causa l'overflow. Fortunatamente usando un cast evita il problema:

```
i = (long) j * j;
```

Dato che l'operatore di cast ha precedenza rispetto all'operatore \*, la variabile j viene prima convertita al tipo long, forzando così la seconda j a essere convertita a sua volta.

Osservate che l'istruzione

```
i = (long) (j * j); *** SBAGLIATO ***/
```

non funzionerebbe dato che l'overflow avrebbe già avuto luogo al momento del cast.

## 7.5 Definizione di tipi

Nella Sezione 5.2 abbiamo usato la direttiva `#define` per creare un macro che avrebbe potuto essere usata come un tipo Booleano:

```
#define BOOL int
```

**D&R** Un modo migliore per creare un tipo Booleano è quello di usare la funzionalità detta di **definizione di tipo** (*type definition*):

```
typedef int Bool;
```

Osservate come il nome del tipo che deve essere definito viene posto alla *fine*. Notate anche che la parola `Bool` ha la prima lettera maiuscola. Usare una maiuscola come prima lettera non è obbligatorio, è solo una convenzione che viene utilizzata da alcuni programmatori.

Usare `typedef` per definire il tipo `Bool` fa sì che il compilatore aggiunga `Bool` alla lista dei nomi di tipi che è in grado di riconoscere. Adesso `Bool` può essere utilizzato nello stesso modo dei tipi nativi, ovvero nelle dichiarazioni di variabili, nelle espressioni di cast e in qualsiasi altro punto. Per esempio possiamo usare `Bool` per dichiarare delle variabili:

```
Bool flag; /* equivale a scrivere int flag */
```

Il compilatore tratta `Bool` come un sinonimo di `int` e quindi la `flag` non è altro che una normale variabile di tipo `int`.

### Vantaggi della definizione di tipi

Le definizioni di tipo possono rendere i programmi più comprensibili (assumendo che il programmatore si dimostri accorto scegliendo nomi che abbiano un certo significato). Supponete per esempio che le variabili `cash_in` e `cash_out` vengano usate per memorizzare delle somme in dollari. Dichiarare `Dollars` come

```
typedef float Dollars;
```

e poi scrivere

```
Dollars cash_in, cash_out;
```

è sicuramente più efficace di

```
float cash_in, cash_out;
```

Le definizioni di tipo inoltre rendono un programma più facile da modificare. Se successivamente decidessimo che `Dollars` debba essere definito come un `double`, tutto quello che dovremmo fare è semplicemente modificare la definizione del tipo:

```
typedef double Dollars;
```

Le dichiarazioni delle variabili `Dollars` non avrebbero bisogno di essere cambiate. Senza la definizione di tipi avremmo avuto bisogno di cercare tutte le variabili `float` usate per memorizzare somme in dollari (non è necessariamente un compito semplice) e cambiare le dichiarazioni.

## Definizione di tipi e portabilità

Le definizioni di tipi sono uno strumento importante per scrivere programmi portabili. Uno dei problemi nel trasferire programmi da un computer all'altro su macchine di tipo diverso i tipi possono presentare intervalli differenti. Se i valori di una variabile `int`, un assegnamento come

```
i = 100000;
```

è corretta su una macchina a 32 bit, mentre non andrebbe a buon fine su una macchina con interi a 16 bit.

### PORATIBILITÀ

*Per una maggiore portabilità considerate la possibilità di usare `typedef` per definire nuovi tipi.*

Supponete di dover scrivere un programma che necessita di variabili in grado di memorizzare le quantità di prodotto nell'intervallo 0-50,000. A questo scopo potremmo usare variabili `long` (in quanto garantiscono di poter contenere numeri compresi tra -2,147,483,647 e +2,147,483,647), tuttavia preferiamo usare variabili `int` perché le operazioni aritmetiche su queste ultime sono più veloci rispetto a quelle sulle variabili `long`. Inoltre a questo le variabili `int` richiedono meno spazio.

Invece di usare il tipo `int` per dichiarare le variabili quantità, possiamo definire un nostro tipo "quantità":

```
typedef int Quantity;
```

e usare questo tipo per dichiarare le variabili:

```
Quantity q;
```

Quando trasferiamo il programma su una macchina in cui gli interi sono più piccoli, possiamo cambiare la definizione di `Quantity`:

```
typedef long Quantity;
```

Sfortunatamente questa tecnica non risolve tutti i problemi considerato il fatto che la modifica alla definizione di `Quantity` non può avere effetto sul modo in cui le variabili `Quantity` vengono usate. Come minimo devono essere modificate tutte le chiamate a `printf` e a `scanf` che usano variabili di tipo `Quantity`, rimpiazzando la conversione `%d` con la `%ld`.

La stessa libreria C usa `typedef` per dichiarare dei nomi per i tipi che possono variare da un'implementazione del C a un'altra. Questi tipi spesso hanno dei nomi che finiscono per `_t`, come `ptrdiff_t`, `size_t` e `wchar_t`. La definizione esatta di questi tipi può variare, ma qui vengono riportati degli esempi tipici:

```
typedef long int ptrdiff_t;
typedef unsigned long int size_t;
typedef int wchar_t;
```

C99

Nel C99 l'header `<stdint.h>` [header `<stdint.h>` 27.1] usa `typedef` per definire i nomi dei vari tipi di interi, associando un particolare numero di bit. Per esempio `int32_t` è un intero con segno di esattamente 32 bit. Usare questi tipi è un modo efficace per scrivere programmi più portabili.

## 7.6 L'operatore sizeof

L'operatore `sizeof` permette a un programma di determinare quanta memoria viene richiesta per memorizzare un valore di un particolare tipo. Il valore dell'espressione

`sizeof( nome-del-tipo )`

D&R

è un intero senza segno che rappresenta il numero di byte richiesti per memorizzare un valore appartenente al tipo *nome-del-tipo*. Il valore `sizeof(char)` è sempre 1, ma la dimensione degli altri tipi può variare. Su una macchina a 32 bit `sizeof(int)` è normalmente uguale a 4. Osservate che `sizeof` è un operatore piuttosto inusuale dato che tipicamente è il compilatore stesso a determinare il valore dell'espressione `sizeof`.

L'operatore `sizeof` può essere applicato anche alle costanti, alle variabili e alle espressioni in generale. Se *i* e *j* sono delle variabili `int`, allora su una macchina a 32 bit `sizeof(i)` è pari a 4, così come lo è `sizeof(i + j)`. Quando `sizeof` viene applicato a un'espressione (invece che a un tipo) non richiede parentesi. Possiamo scrivere `sizeof i` invece che `sizeof(i)`. In ogni caso le parentesi potrebbero essere comunque necessarie a causa dell'ordine di precedenza. Il compilatore interpreta `sizeof i + j` come `sizeof(i) + j` a causa del fatto che `sizeof` (che è un operatore unario) ha precedenza sull'operatore binario `+`. Per evitare problemi è meglio usare le parentesi in tutte le espressioni `sizeof`.

Stampare un valore `sizeof` richiede un po' di attenzione a causa del fatto che il tipo di un'espressione `sizeof` è `size_t` e questo viene definito dall'implementazione. Nel C89 la cosa migliore è convertire il valore dell'espressione in un tipo conosciuto prima di stamparlo. Viene garantito che il `size_t` sia di tipo `unsigned` e quindi la conversione più sicura è quella di fare un cast dell'espressione `sizeof` nel tipo `unsigned long` (il più grande dei tipi `unsigned` del C89) e poi stamparla usando la conversione `%lu`:

```
printf("Size of int: %lu\n", (unsigned long) sizeof(int));
```

C99:

Nel C99 il tipo `size_t` può essere più grande di un `unsigned long`. Tuttavia la funzione `printf` del C99 è in grado di visualizzare direttamente i valori `size_t` senza la necessità di eseguire un cast. Il trucco è quello di usare nella specifica di conversione la lettera `z` seguita da uno dei soliti codici per gli interi (tipicamente `u`):

```
printf("Size of int: %zu\n", sizeof(int)); /* solo C99 */
```

## Domande & Risposte

**D:** Nella Sezione 7.1 viene detto che le specifiche `%o` e `%x` sono usate per scrivere interi senza segno nella notazione ottale ed esadecimale. Come è possibile scrivere i normali interi con segno nei formati ottale ed esadecimale? [p.136]

**R:** Potete usare `%o` e `%x` per stampare un intero con segno ammesso che il valore di questo non sia negativo. Queste conversioni fanno sì che la `printf` tratti un intero con segno come se fosse un intero senza segno. In altre parole la `printf` assume che il bit di segno faccia parte del valore assoluto del numero. Fintanto che il bit di segno è uguale

a 0 non ci sono problemi. Se il bit di segno è uguale a 1 allora la printf stamperà un numero insolitamente grande.

**D:** Ma cosa succede se il numero è negativo? Come possiamo scriverlo in ottale o esadecimale?

**R:** Non c'è un modo diretto per stampare in ottale o esadecimale un numero negativo. Fortunatamente la necessità di farlo è piuttosto rara. Potete naturalmente controllare se il numero è negativo e stampare voi stessi un segno meno:

```
if (i < 0)
 printf("-%x", -i);
else
 printf("%x", i);
```

**D:** Perché le costanti floating point vengono memorizzate nel formato double invece che in quello float? [p. 139]

**R:** Per ragioni storiche il C dà preferenza al tipo double, mentre quello float è considerato un cittadino di seconda classe. Considerate per esempio la discussione sui float nel libro *The C Programming Language* di Kernighan e Ritchie: "La ragione principale per utilizzare il tipo float è quello di risparmiare dello spazio nei vettori di grandi dimensioni, oppure, più raramente, per risparmiare tempo su macchine dove l'aritmetica a doppia precisione è particolarmente onerosa." Originariamente il C imponeva che tutte le operazioni aritmetiche in floating point venissero fatte in doppia precisione (il C89 e il C99 non hanno quest'obbligo).

**\*D:** Come sono fatte e a cosa servono le costanti a virgola mobile esadecimali? [p. 140]

**R:** Una costante a virgola mobile esadecimale comincia per 0x o 0X e deve contenere un esponente che è preceduto dalla lettera P (o p). L'esponente può avere un segno e la costante può finire per f, F, l o L. L'esponente è espresso in formato decimale ma rappresenta una potenza di 2 e non una potenza di 10. Per esempio, 0x1.Bp3 rappresenta il numero  $1.6875 \times 2^3 = 13.5$ . La cifra esadecimale B corrisponde al pattern dei bit 1011. La B si trova a destra del punto e quindi ogni bit a 1 rappresenta una potenza negativa di 2. Sommando queste potenze di 2 ( $2^{-1} + 2^{-3} + 2^{-4}$ ) si ottiene 0.6875.

Le costanti a virgola mobile esadecimali sono utili principalmente per specificare costanti che richiedono una grande precisione (incluse le costanti matematiche come e e  $\pi$ ). I numeri esadecimali hanno una rappresentazione binaria ben precisa, una costante scritta nel formato decimale invece è soggetta a piccoli errori di arrotondamento quando viene convertita in decimale. I numeri esadecimali sono utili anche per definire costanti dei valori estremi, come quelli delle macro presenti nell'header <float.h>. Queste costanti sono facili da scrivere in esadecimale mentre sono difficili da esprimere in decimale.

**\*D:** Perché per leggere i double viene usata la specifica %lf mentre per stamparli usiamo il %f? [p. 140]

**R:** Questa è una domanda cui è difficile rispondere. Per prima cosa tenete presente che la scanf e la printf sono delle funzioni inusuali perché non sono costrette a avere un numero prefissato di argomenti. Possiamo dire che la scanf e la printf hanno una

lista di argomenti di lunghezza variabile [[lista di argomenti di lunghezza variabile > 26.1](#)]. Quando funzioni con una lista di argomenti di lunghezza variabile vengono chiamate, il compilatore fa sì che gli argomenti float vengano convertiti al tipo double. Come risultato la printf non è in grado di distinguere tra argomenti float e argomenti double. Questo spiega perché %f funziona sia per gli argomenti di tipo float che per quelli di tipo double nelle chiamate alla printf.

Alla scanf invece viene passato un *puntatore* alla variabile. La specifica %f dice alla scanf di memorizzare un valore float all'indirizzo che le viene passato, mentre la specifica %lf dice alla scanf di memorizzare in quell'indirizzo un valore di tipo double. Qui la differenza tra float e double è essenziale. Se viene fornita la specifica di conversione sbagliata, la scanf memorizzerà un numero errato di byte (senza menzionare il fatto che lo schema dei bit di un float è diverso da quello di un double).

**D: Qual è il modo corretto per pronunciare char? [p. 140]**

R: Non c'è una pronuncia universalmente accettata. Alcune persone pronunciano *char* allo stesso modo in cui si pronuncia la prima sillaba della parola "character" ('kær kt (r) nell'alfabeto fonetico internazionale). Altri dicono utilizzano la pronuncia di "char broiled" (t (r) nell'alfabeto fonetico internazionale).

**D: In quali casi ha importanza se una variabile char è di tipo signed o unsigned? [p. 142]**

R: Se nella variabile memorizziamo solo caratteri a 7 bit, allora non ha nessuna importanza dato che il bit di segno sarà uguale a 0. Se invece pianifichiamo di salvare caratteri a 8 bit, allora probabilmente vorremo che la variabile sia di tipo unsigned char. Considerate l'esempio seguente:

```
ch = '\xdb';
```

Se ch è stata dichiarata di tipo char, allora il compilatore può decidere di trattarla come un carattere con segno (molti compilatori lo fanno). Fintanto che ch viene usata come un carattere allora non avremo problemi. Tuttavia se ch fosse usata in un contesto in cui viene richiesto al compilatore di convertire il suo valore in un intero, allora probabilmente si presenterà un problema: l'intero risultante sarà negativo dato che il bit di segno di ch è uguale a 1.

Ecco un'altra situazione: in certi tipi di programmi, è consuetudine memorizzare interi composti da un singolo byte all'intero di variabili char. Se stessimo scrivendo un programma di questo tipo, allora dovremmo decidere se ogni variabile debba essere signed char o unsigned char, così come per le variabili intere ordinarie decidiamo se debbano essere di tipo int o unsigned int.

**D: Non capiamo come il carattere new-line possa essere il carattere ASCII line-feed. Quando un utente immette l'input e pigia il tasto Invio, il programma non dovrebbe leggere un carattere di carriage-return oppure un carriage-return seguito da un carattere line-feed? [p.143]**

R: No. Per eredità dallo UNIX, il C considera sempre la fine di una riga come delimitata da un singolo carattere di line-feed (in UNIX, nei file testuali alla fine di una riga appare un carattere line-feed e nessun carattere carriage-return). La libreria del C si prende cura di tradurre il tasto premuto dall'utente in un carattere line-feed.

Quando un programma legge da file, la libreria di I/O traduce il delimitatore end-of-line (qualsiasi esso sia) in un singolo carattere line-feed. La medesima trasformazione avviene (nel senso opposto) quando l'output viene scritto a video o su un file (Si veda la Sezione 22.1 per i dettagli).

Sebbene queste trasformazioni possano sembrare un motivo di confusione, hanno uno scopo importante: isolare i programmi dai dettagli che possono variare da un sistema operativo all'altro.

**\*D: Qual è lo scopo della sequenza di escape \? ? [p.143]**

**R:** La sequenza di escape è legata alle sequenze trigrafiche [sequenze trigrafiche > 25.3] che iniziano per ???. Se avete bisogno di inserire un ??? in una stringa, c'è la possibilità che il compilatore la scambi per l'inizio di una sequenza trigrafica. Rimettere il secondo ? con un \? risolve il problema.

**D: Se getchar è più veloce perché dovremmo voler usare la scanf per leggere dei caratteri individuali? [p. 146]**

**R:** Sebbene non sia veloce come la getchar, la funzione scanf è più flessibile. Come abbiamo dato precedentemente la stringa di formato "%c" fa sì che la scanf legga il prossimo carattere di input, mentre "%c" in modo che venga letto il successivo carattere che non sia uno spazio bianco. Inoltre la scanf è efficace nella lettura di caratteri che sono mischiati con altri tipi di dati. Diciamo per esempio che l'input sia costituito da un intero seguito da un singolo carattere non numerico e infine un altro intero. Usando nella scanf la stringa di formato "%d%c%d" possiamo leggere tutti e tre gli oggetti.

**\*D: In quali circostanze le promozioni integrali convertono un carattere o un intero short in un unsigned int? [p. 149]**

**R:** Le promozioni integrali restituiscono un unsigned int nel caso in cui il tipo int non sia sufficientemente grande da includere tutti i possibili valori contenuti dal tipo originale. Dato che i caratteri di solito sono lunghi 8 bit, sono quasi sempre convertiti in un int che garantisce di essere lungo almeno 16 bit. Allo stesso modo anche gli interi short possono essere sempre convertiti in un int. Gli unsigned short integer sono problematici. Se gli interi short hanno la stessa lunghezza dei normali interi (come accade nelle macchine a 16 bit), allora gli interi unsigned short devono essere convertiti nel tipo unsigned int, dato che il più grande intero unsigned short (65,535 su macchine a 16 bit) è maggiore del più grande int (32,767).

**D: Cosa accade esattamente quando si assegna un valore a una variabile che non è abbastanza grande per contenerlo? [p. 152]**

**R:** In breve, se il valore è di un integral type e la variabile è di tipo unsigned, allora i bit eccedenti vengono scartati. Se la variabile è di tipo signed allora il risultato dipende dall'implementazione. Assegnare un numero a virgola mobile a una variabile (intera o a virgola mobile) che è troppo piccola per contenerlo produce un comportamento non definito: può succedere qualsiasi cosa, inclusa la terminazione del programma.

**\*D: Perché il C si preoccupa di fornire le definizioni di tipo? Definire Bool come una macro non è una soluzione altrettanto valida che definire un tipo Bool con typedef? [p. 155]**

**R:** Ci sono due differenze importanti tra le definizioni di tipo e le definizioni di macro. Per prima cosa le definizioni di tipo sono più potenti di quelle di macro. In particolare i tipi vettore e i tipi puntatore non possono essere definiti come macro. Supponete di dover usare una macro per definire un tipo "puntatore a intero":

```
#define PTR_TO_INT int *
```

La dichiarazione

```
PTR_TO_INT p, q, r;
```

dopo il preprocessing diventerebbe

```
int * p, q, r;
```

Sfortunatamente solo p è un puntatore mentre q ed r sono delle variabili intere ordinarie. Le definizioni di tipo non soffrono di questi problemi.

Secondariamente i nomi `typedef` non sono soggetti alle stesse regole di *scope* delle variabili. Un nome definito con `typedef` all'intero del corpo di una funzione non verrebbe riconosciuto al di fuori della funzione. I nomi macro invece vengono rimpiattati dal preprocessore in ogni punto in cui appaiono.

**\*D:** Si è detto che il compilatore "solitamente può determinare il valore di un'espressione `sizeof`". Il compilatore non può farlo sempre? [p. 157]

**R:** Nel C89 sì. Nel C99 però c'è un'eccezione. Il compilatore non può determinare la dimensione di un vettore di lunghezza variabile [**vettori a lunghezza variabile > 8.3**] a causa del fatto che il numero degli elementi presenti nel vettore può cambiare durante l'esecuzione del programma.

## Esercizi

### Sezione 7.1

1. Fornite il valore decimale di ognuna delle seguenti costanti intere.

- (a) 077
- (b) 0x77
- (c) 0XABC

### Sezione 7.2

2. Quale delle seguenti costanti non è ammessa dal C? Classificate ogni costante come intera o a virgola mobile.

- (a) 010E2
- (b) 32.1E+5
- (c) 0790
- (d) 100\_000
- (e) 3.978e-2

W 3. Quale dei seguenti non è un tipo ammesso dal C?

- (a) short unsigned int
- (b) short float
- (c) long double
- (d) unsigned long

- Sezione 7.3**
4. Se *c* è una variabile char, quale delle seguenti istruzioni non è ammessa?
- (W)
- (a) *i* += *c*; /\* *i* è di tipo int \*/
  - (b) *c* = 2 \* *c* - 1;
  - (c) putchar(*c*);
  - (d) printf(*c*);
5. Quale fra i seguenti non è un modo corretto per scrivere il numero 65? (Assumete che il set dei caratteri sia ASCII)
- (a) 'A'
  - (b) 0b1000001
  - (c) 0101
  - (d) 0x41
6. Per ognuna delle seguenti tipologie di dato, specificate quale tra char, short e long è il tipo più piccolo che è in grado di garantire di essere grande a sufficienza per memorizzare il dato.
- (a) Giorni in un mese
  - (b) Giorni in un anno
  - (c) Minuti in un giorno
  - (d) Secondi in un giorno
7. Fornite per ciascuno dei seguenti caratteri di escape il codice ottale equivalente. (Assumete che il set di caratteri sia l'ASCII.) Potete consultare l'Appendice A per elenca i codici numerici per i caratteri ASCII.
- (a) \b
  - (b) \n
  - (c) \r
  - (d) \t
8. Ripetete l'Esercizio 7 fornendo il codice di escape equivalente espresso in esadecimale.
- Sezione 7.4**
9. Supponete che *i* e *j* siano delle variabili di tipo int. Qual è il tipo dell'espressione *i* / *j* + 'a'?
- (W)
10. Supponete che *i* sia un variabile di tipo int, *j* una variabile di tipo long e la variabile *k* sia di tipo unsigned int. Qual è il tipo dell'espressione *i* + (*int*)*j* + *k*?
11. Supponete che *i* sia una variabile di tipo int, *f* una variabile di tipo float e la variabile *d* sia di tipo double. Qual è il tipo dell'espressione *i* \* *f* / *d*?
- (W)
12. Supponete che *i* sia un variabile di tipo int, *f* una variabile di tipo float e la variabile *d* sia di tipo double. Spiegate quali conversioni hanno luogo durante l'esecuzione della seguente istruzione:

*d* = *i* + *f*;

13. Assumete che il programma contenga le seguenti dichiarazioni:

```
char c = '\1';
short s = 2;
int i = -3;
long m = 5;
float f = 6.5f;
double d = 7.5;
```

Fornite il valore e il tipo di ognuna delle espressioni qui di seguito elencate:

- (a)  $c * i$
- (c)  $f / c$
- (e)  $f - d$
- (b)  $s + m$
- (d)  $d / s$
- (f)  $\text{int}(f)$

- W 14. Le seguenti istruzioni calcolano sempre in modo corretto la parte frazionaria di  $f$ ? (Assumete che  $f$  e `frac_part` siano variabili `float`.)

```
frac_part = f - (int) f;
```

Se non fosse così, qual è il problema?

- Sezione 7.5 15. Utilizzare `typedef` per creare dei tipi chiamati `Int8`, `Int16` e `Int32`. Definite questi tipi in modo che sulla vostra macchina rappresentino interi a 8, 16 e 32 bit.

## Progetti di programmazione

- W 1. Il programma `square2.c` della Sezione 6.3 non funzionerà (tipicamente stamperebbe delle risposte strane) se  $i * i$  eccede il massimo valore `int`. Fate girare il programma e determinate il più piccolo valore di  $n$  che causa il problema. Provate a cambiare il tipo di  $i$  nel tipo `short` ed eseguite nuovamente il programma (non dimenticatevi di aggiornare la specifica di conversione nella chiamata alla `printf`). Successivamente provate con il tipo `long`. Cosa potete concludere da questi esperimenti sul numero di bit usati per memorizzare nella vostra macchina diversi tipi interi?
- W 2. Modificate il programma `square2.c` della Sezione 6.3 in modo che faccia una pausa ogni 24 quadrati e visualizzi il seguente messaggio:
- Press Enter to continue...
- Dopo aver visualizzato il messaggio, il programma deve usare `getchar` per leggere un carattere. La funzione `getchar` non permetterà al programma di proseguire fino a quando l'utente non avrà pigliato il tasto Invio.
3. Modificate il programma `sum2.c` della Sezione 7.1 per sommare una serie di numeri `double`.
4. Scrivete un programma che traduca il numero telefonico alfabetico nella sua forma numerica:
- Enter phone number: CALLATT  
2255288

(Nel caso in cui non avete un telefono nelle vicinanze, queste sono le letture:  
tasti: 2=ABC, 3=DEF, 4=GHI, 5=JKL, 6=MNO, 7=PRS, 8=TUV, 9=WXZ).  
Se il numero di telefono originale contiene caratteri non alfabetici (cifre o simboli di interpunkzione), lasciateli esattamente come sono:

Enter phone number: 1-800-COL-LECT  
1-800-265-5328

Potete assumere che tutte le lettere immesse dall'utente siano maiuscole.

- W 5. Nel gioco dello SCARABEO, i giocatori formano delle parole usando piccole tessere, ognuna contenente una lettera e un valore. I valori di variano da lettera a lettera sulla base della rarità della lettera stessa (i valori delle lettere nella versione inglese del gioco sono: 1:AEILNORSTU, 2:DG, 3:BCP, 4:FHVWY, 5:K, 8:JX, 10:QZ). Scrivete un programma che calcoli il valore di una parola sommando il valore associato alle sue lettere:

Enter a word: pitfall  
Scrabble value: 12

Il vostro programma deve permettere all'interno della parola un miscuglio di lettere minuscole e maiuscole. Suggerimento: usate la funzione di libreria touppercase()

- W 6. Scrivete un programma che stampi i valori sizeof(int), sizeof(short), sizeof(long), sizeof(float), sizeof(double) e sizeof(long double).
- 7. Modificate il Progetto di programmazione 6 del Capitolo 3 in modo che sia in grado di sottrarre, moltiplicare o dividere le due frazioni immettendo +, -, \* o / tra le frazioni stesse.
- 8. Modificate il Progetto di Programmazione del Capitolo 5 in modo che l'utente immetta un orario nel formato a 12 ore. L'input deve avere la forma ore:minuti seguito da A, P, AM o PM (sia in minuscole che maiuscole). Spazi bianchi tra l'ora e l'indicatore AM/PM sono ammessi (ma non necessari). Ecco degli esempi di input validi:

```
1:15P
1:15PM
1:15p
1:15pm
1:15 P
1:15 PM
1:15 p
1:15 pm
```

Potete assumere che l'input abbia una di queste forme, non c'è bisogno di fare un test per rilevare possibili errori.

- 9. Scrivete un programma che chieda all'utente un orario nel formato a 12 ore e lo stampi nel formato a 24 ore:

Enter a 12-hour time: 9:11 PM
Equivalent 24-hour time: 21:11

Guardate il Progetto di programmazione 8 per una descrizione del **fornito** input.

- 10.** Scrivete un programma che conti il numero di vocali in una frase:

Enter a sentence: And that's the way it is.

Your sentence contains 6 vowels.

- 11.** Scrivete un programma che prenda un nome e un cognome immessi e stampi cognome, una virgola e l'iniziale del nome seguita da un punto:

Enter a first and last name: Lloyd Fosdick

Fosdick, L.

L'input immesso dall'utente può contenere degli spazi aggiuntivi prima e dopo il nome e il cognome e dopo il cognome.

- 12.** Scrivete un programma che calcoli un'espressione:

Enter an expression:  $1+2.5*3$

Value of expression: 10.5

Gli operandi dell'espressione sono numeri floating point. Gli operatori sono  $*$  e  $/$ . L'espressione viene calcolata da sinistra a destra (nessun operatore ha precedenza sugli altri).

- 13.** Scrivete un programma che calcoli la lunghezza media delle parole in una frase:

Enter a sentence: It was deja vu all over again.

Average word length: 3.4

Per semplicità il programma deve considerare un segno di interpunkzione e facente parte della parola alla quale è attaccato. Stampate la lunghezza media delle parole con una cifra decimale.

- 14.** Scrivete un programma che usi il metodo di Newton per calcolare la radice quadrata di un numero positivo a virgola mobile:

Enter a positive number: 3

Square root: 1.73205

Sia  $x$  un numero immesso dall'utente. Il metodo di Newton richiede una stima iniziale  $y$  della radice quadrata di  $x$  (noi useremo  $y=1$ ). Le stime successive vengono trovate calcolando la media di  $y$  e  $x/y$ . La tabella seguente illustra come viene trovata la radice quadrata di 3:

| $x$ | $y$     | $x/y$   | Media di $y$ e $x/y$ |
|-----|---------|---------|----------------------|
| 3   | 1       | 3       | 2                    |
| 3   | 2       | 1.5     | 1.75                 |
| 3   | 1.75    | 1.71429 | 1.73214              |
| 3   | 1.73214 | 1.73196 | 1.73205              |
| 3   | 1.73205 | 1.73205 | 1.73205              |

Osservate che i valori di  $y$  diventano progressivamente più vicini alla vera radice di  $x$ . Per una precisione più accurata il vostro programma deve usare variabili di tipo double invece che del tipo float. Il programma deve terminare quando il valore assoluto della differenza tra il vecchio valore di  $y$  e il nuovo valore di  $y$  è minore del prodotto tra 0.00001 e  $y$ . *Suggerimento:* usate la funzione fabs per trovare il valore assoluto di un double (per poter usare la funzione fabs avrete bisogno di includere l'header <math.h> all'inizio del vostro programma).

**15.** Scrivete un programma che calcoli il fattoriale di un numero intero positivo:

Enter a positive integer: 6

Factorial of 6: 720

- (a) Usate una variabile short per salvare il valore del fattoriale. Qual è il più grande numero  $n$  di cui il programma calcola correttamente il fattoriale?
- (b) Ripetete la parte (a) usando una variabile int.
- (c) Ripetete la parte (a) usando una variabile long.
- (d) Ripetete la parte (a) usando una variabile long long (se il vostro compilatore supporta questo tipo).
- (e) Ripetete la parte (a) usando una variabile float.
- (f) Ripetete la parte (a) usando una variabile double.
- (g) Ripetete la parte (a) usando una variabile long double.

Nei casi dalla (e) alla (g) il programma visualizzerà un'approssimazione del fattoriale, non necessariamente il valore esatto.

# 8 Vettori

Finora abbiamo visto solo variabili **scalari**, cioè capaci di contenere dati costituiti da un singolo elemento. Il C supporta anche variabili **aggregate** che sono in grado di memorizzare delle collezioni di valori. Nel C ci sono due tipi di variabili aggregate: i vettori e le strutture. Questo capitolo illustra come dichiarare e usare vettori sia di tipo unidimensionale (Sezione 8.1) che multidimensionale (Sezione 8.2). La Sezione 8.3 tratta i vettori a lunghezza variabile dello standard C99. Il capitolo è focalizzato principalmente sui vettori unidimensionali, i quali giocano un ruolo molto più importante di quelli multidimensionali all'interno della programmazione C. I capitoli successivi (il Capitolo 12 in particolare) forniranno delle informazioni aggiuntive sui vettori. Il Capitolo 16 invece si occuperà delle strutture.

## 8.1 Vettori unidimensionali

Un **vettore** (*array*) è una struttura contenente un certo numero di dati, tutti dello stesso tipo. Questi valori, chiamati **elementi**, possono essere selezionati individualmente tramite la loro posizione all'interno del vettore.

Il tipo più semplice di vettore ha una sola dimensione. Gli elementi di un vettore unidimensionale sono concettualmente disposti uno dopo l'altro su una riga (o colonna se preferite). Ecco come si potrebbe visualizzare un vettore unidimensionale chiamato *a*:



Per dichiarare un vettore dobbiamo specificare il *tipo* e il *numero* dei suoi elementi. Per esempio per dichiarare che il vettore *a* è costituito da 10 elementi di tipo *int* dobbiamo scrivere:

```
int a[10];
```

Gli elementi di un vettore possono essere di qualsiasi tipo e la sua lunghezza può essere specificata da una qualsiasi espressione (intera) costante [**espressioni intere > 5.3**].

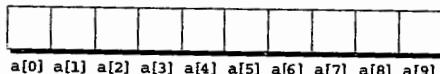
Considerato che in una versione successiva del programma la lunghezza del vettore potrebbe dover essere modificata, è preferibile definirla tramite una macro:

```
#define N 10
```

```
-
int a[N];
```

## Indicizzazione di un vettore

Per accedere a un particolare elemento di un vettore dobbiamo scrivere il nome del vettore seguito da un valore intero racchiuso tra parentesi quadre (questa operazione viene chiamata **indicizzazione** o **subscripting** del vettore). Gli elementi di un vettore vengono sempre contati a partire dallo 0 e quindi in un vettore di lunghezza  $n$  hanno un indirizzo che va da 0 a  $n-1$ . Per esempio, se il vettore a contiene 10 elementi, questi sono identificati come  $a[0]$ ,  $a[1]$ , ...,  $a[9]$ , così come illustrato dalla seguente figura:



Espressioni della forma  $a[i]$  sono degli lvalue [**lvalue > 4.2**] e quindi possono essere usati come delle normali variabili:

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```

In generale, se un vettore contiene elementi del tipo  $T$ , allora ogni elemento viene trattato come se fosse una variabile di tipo  $T$ . In questo esempio, gli elementi  $a[0]$ ,  $a[5]$  e  $a[i]$  si comportano come variabili di tipo int.

I vettori e i cicli for sono spesso abbinati. Molti programmi contengono cicli for il cui unico scopo è quello di effettuare la stessa operazione su ciascun elemento del vettore. Ecco alcuni esempi di tipiche operazioni effettuabili su un vettore a di lunghezza N:

```
for (i = 0; i < N; i++)
 a[i] = 0; /* azzerà gli elementi di a */

for (i = 0; i < N; i++)
 scanf("%d", &a[i]); /* legge dei dati e li mette in a */

for (i = 0; i < N; i++)
 sum += a[i]; /* somma gli elementi di a */
```

Osservate che, quando viene chiamata la scanf per leggere un elemento da mettere nel vettore, dobbiamo inserire il simbolo & esattamente come avremmo fatto per una normale variabile.



Il C non richiede che i limiti di un vettore vengano controllati mentre vi si accede. Se un indice va fuori dall'intervallo ammesso per il vettore, il comportamento del programma non è definito. Una delle cause che portano un indice a oltrepassare i limiti è dimenticare che per un vettore di  $n$  elementi gli indici vanno da 0 a  $n-1$  e non da 1 a  $n$ . L'esempio seguente illustra lo strano effetto che può essere causato da questo errore comune:

```
int a[10], i;
for (i = 1; i <= 10; i++)
 a[i] = 0;
```

Con alcuni compilatori questo "innocente" ciclo `for` può causare un ciclo infinito! Quando `i` raggiunge il valore 10, il programma memorizza uno zero in `a[10]`. Ma `a[10]` non esiste e quindi lo 0 viene messo nella memoria immediatamente dopo `a[9]`. Se nella memoria la variabile `i` viene a trovarsi dopo `a[9]` (come dovrebbe accadere in questo caso) allora `i` viene imposta a 0 facendo sì che il ciclo abbia nuovamente inizio.

L'indice di un vettore può essere costituito da un'espressione intera:

```
a[i+j*10] = 0;
```

L'espressione può avere dei side effect:

```
i = 0;
while (i < N)
 a[i++] = 0;
```

Tracciamo l'esecuzione di questo codice. Dopo che `i` viene impostata a 0, l'istruzione `while` controlla se `i` è minore di `N`. In tal caso, ad `a[0]` viene assegnato uno 0, `i` viene incrementato e il ciclo si ripete. Fate attenzione al fatto che `a[+i]` non sarebbe corretto visto che, durante la prima iterazione del ciclo, ad `a[1]` verrebbe assegnato il valore 0.



Fate attenzione quando, indicizzando un vettore, si hanno degli effetti secondari. Il ciclo seguente per esempio (che si suppone faccia la copia degli elementi dal vettore `b` al vettore `a`) potrebbe non funzionare a dovere:

```
i = 0;
while (i < N)
 a[i] = b[i++];
```

L'espressione `a[i] = b[i++]` oltre ad accedere al valore di `i`, lo modifica in un altro punto dell'espressione stessa, il che, come abbiamo visto nella Sezione 4.4, provoca un comportamento indefinito. Naturalmente possiamo evitare facilmente il problema rimuovendo l'incremento dall'indicizzazione:

```
for (i = 0; i < N, i++)
 a[i] = b[i];
```

PRIMAVERA

## Invertire una serie di numeri

Il nostro primo programma sui vettori chiede all'utente di immettere un serie di numeri e poi li riscrive in ordine inverso:

Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31

In reverse order: 31 50 11 23 94 7 102 49 82 34

La nostra strategia sarà quella di salvare i numeri letti all'intero di un vettore e poi procedere a ritroso nel vettore stesso stampando i suoi elementi uno a uno. In altre parole non invertiamo l'ordine dei numeri all'interno del vettore, lo facciamo solo credere all'utente.

```
/* Inverte una serie di numeri */

#include <stdio.h>

#define N 10

int main(void)
{
 int a[N], i;

 printf("Enter %d numbers: ", N);
 for (i = 0; i < N; i++)
 scanf("%d", &a[i]);

 printf("In reverse order:");
 for (i = N - 1; i >= 0; i--)
 printf(" %d", a[i]);
 printf("\n");

 return 0;
}
```

Questo programma dimostra quanto siano utili le macro utilizzate congiuntamente ai vettori. La macro N viene usata quattro volte all'interno del programma: nella dichiarazione di a, nella printf che visualizza la richiesta all'utente e in entrambi i cicli for. Se in un secondo momento dovessimo decidere di cambiare la dimensione del vettore, dovremmo solo modificare la definizione di N e ricompilare il programma. Non verrebbe cambiato nient'altro, persino il messaggio per l'utente sarebbe ancora corretto.

## Inizializzazione dei vettori

A un vettore, come a ogni altra variabile, può venir assegnato un valore iniziale al momento della dichiarazione. Le regole sono in qualche modo complicate, tuttavia ne vedremo alcune ora, mentre tratteremo le altre più avanti [inizializzatori > 18.5].

La forma più comune di **inizializzatore** per un vettore è una lista di espressioni connesse racchiuse tra parentesi graffe e separate da virgole:

int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

Se l'inizializzatore è più corto del vettore, allora agli elementi restanti del vettore viene imposto il valore zero:

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* il valore iniziale è {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

Sfruttando questa caratteristica possiamo inizializzare a zero un vettore in modo molto semplice:

```
int a[10] = {0};
/* il valore iniziale è {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

Non è ammesso che un inizializzatore sia completamente vuoto, per questo mettiamo un singolo 0 all'interno delle parentesi graffe. A un inizializzatore non è ammesso neanche di avere una lunghezza maggiore di quella del vettore.

Se è presente un inizializzatore, la lunghezza del vettore può essere omessa:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Il compilatore usa la lunghezza dell'inizializzatore per determinare quale sia la lunghezza del vettore. Il vettore ha comunque una lunghezza fissa (pari a 10 in questo caso) proprio come se avessimo specificato la lunghezza in modo esplicito.

## Designatori inizializzati

Accade spesso che solo pochi elementi di un vettore debbano essere inizializzati esplicitamente lasciando agli altri il valore di default. Considerate il seguente esempio:

```
int a[15] = {0, 0, 29, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 48};
```

Vogliamo che l'elemento 2 del vettore sia 29, che l'elemento 9 sia uguale a 7 e che l'elemento 14 sia uguale 48, mentre gli altri valori saranno imposti semplicemente a zero. Scrivere inizializzatori di questo tipo per vettori di dimensioni considerevoli è tedioso e potenziale fonte di errori (che succederebbe se tra due valori diversi da zero ci fossero 200 zeri?)

I **designatori inizializzati** del C99 possono essere usati per risolvere questo problema. Ecco come dovremmo riscrivere l'esempio precedente usando questo tipo di inizializzatori:

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

Ogni numero tra parentesi quadre viene detto **designatore**.

Oltre a essere brevi e più facili da leggere (almeno per certi vettori), i designatori inizializzati hanno un altro vantaggio: l'ordine in cui vengono elencati gli elementi non ha alcuna importanza. Quindi l'esempio precedente poteva essere scritto anche in questo modo:

```
int a[15] = {[14] = 48, [9] = 7, [2] = 29};
```

I designatori devono essere delle espressioni intere costanti. Se il vettore che deve essere inizializzato è lungo  $n$ , allora ogni designatore deve essere compreso tra 0 e  $n-1$ . Tuttavia se la lunghezza del vettore viene omessa, un designatore può essere un

qualsiasi intero non negativo. In quest'ultimo caso il compilatore dedurrà la lunghezza del vettore dal designatore più grande.

Nell'esempio seguente il fatto che il 23 appaia come un designatore fa sì che la lunghezza del vettore sia 24:

```
int b[] = {[5] = 10, [23] = 13, [11] = 36, [15] = 29};
```

Un inizializzatore può usare contemporaneamente sia la tecnica vecchia (elemento per elemento) che quella nuova (indicizzata):

```
int c[10] = {[5, 1, 9, [4] = 3, 7, 2, [8] = 6];
```



L'inizializzatore specifica che i primi tre elementi debbano essere un 5, un 1 e un 9. L'elemento 4 deve avere valore 3. I due elementi dopo l'elemento 4 devono avere i valori 7 e 2. Infine l'elemento 8 deve essere uguale a 6. Tutti gli elementi per i quali non è specificato alcun valore saranno uguali a 0 per default.

#### PROGRAMMA

## Controllare il numero di cifre ripetute

Il nostro prossimo programma controlla se una delle cifre presenti in un numero appare più di una volta all'interno del numero stesso. Dopo l'immissione del numero da parte dell'utente, il programma stampa il messaggio Repeated digit o il messaggio No repeated digit:

Enter a number: 28212

Repeated digit

Il numero 28212 ha un cifra ripetuta (il 2), mentre un numero come 9357 non ne ha.

Il programma usa un vettore di valori booleani per tenere traccia delle cifre presenti nel numero. Il vettore, chiamato `digit_seen`, ha indici che vanno da 0 a 9 corrispondenti alle 10 possibili cifre. Inizialmente ogni elemento del vettore è falso (l'inizializzatore per `digit_seen` è `{false}`, e inizializza solo il primo elemento del vettore, rendendo gli altri valori uguali a 0, che è del tutto equivalente al valore `false`).

Quando viene dato il numero `n`, il programma lo esamina una cifra alla volta salvando questa nella variabile `digit` e poi usandola come indice per `digit_seen`. Se `digit_seen[digit]` è vero questo significa che la cifra `digit` è contenuta almeno due volte all'interno di `n`. D'altra parte se `digit_seen[digit]` è falso, allora la cifra `digit` non è mai stata vista prima e quindi il programma impone `digit_seen[digit]` al valore `true` e continua l'esecuzione.

```
repdigitc /* Controlla se un numero ha delle cifre ripetute */

#include <stdbool.h> /* solo C99 */
#include <stdio.h>

int main(void)
{
 bool digit_seen[10] = {false};
 int digit;
 long n;
```

```

printf("Enter a number: ");
scanf("%ld", &n);
while (n > 0) {
 digit = n % 10;
 if (digit_seen[digit])
 break;
 digit_seen[digit] = true;
 n /= 10;
}
if (n > 0)
 printf("Repeated digit\n");
else
 printf("No repeated digit\n");
return 0;
}

```



Questo programma usa i nomi `bool`, `true` e `false` che sono definiti nell'header `<stdbool.h>` del C99 [header `<stdbool.h>` > 21.5]. Se il vostro compilatore non supporta questo header, allora dovrete definire questi nomi voi stessi. Un modo per farlo è quello di inserire queste linee sopra la funzione `main`:

```

#define true 1
#define false 0
typedef int bool;

```

Osservate che `n` è di tipo `long` e questo permette all'utente di immettere numeri fino a 2,147,483,647 (o più, in alcune macchine).

## Usare l'operatore `sizeof` con i vettori

L'operatore `sizeof` può determinare la dimensione di un vettore (in byte). Se `a` è un vettore di 10 interi, allora tipicamente `sizeof(a)` sarà uguale a 40 (assumendo che ogni intero richieda quattro byte).

Possiamo usare `sizeof` anche per misurare la dimensione di un elemento di un vettore come `a[0]`. Dividendo la dimensione del vettore per la dimensione di un elemento si ottiene la lunghezza del vettore:

```
sizeof(a) / sizeof(a[0])
```

Alcuni programmati utilizzano questa espressione quando è necessario ricavare la dimensione di un vettore. Per esempio, per azzerare il vettore `a`, possiamo scrivere

```
for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)
 a[i] = 0;
```

Con questa tecnica il ciclo non deve essere modificato nel caso la lunghezza del vettore venisse modificata in un secondo momento. Usare una macro che rappresenti la lunghezza del vettore ha gli stessi vantaggi naturalmente, ma la tecnica `sizeof` è leggermente migliore dato che non occorre ricordare nessun nome di macro.

Con alcuni compilatori si riscontra un piccolo inconveniente perché questi producono un messaggio di warning per l'espressione `i < sizeof(a) / sizeof(a[0])`. La variabile `i` probabilmente è di tipo `int` (un tipo con segno) mentre `sizeof` produce un valore di tipo `size_t` (un tipo senza segno). Sappiamo dalla Sezione 7.4 che confrontare un intero `signed` con un intero `unsigned` è una pratica pericolosa sebbene in questo caso sia sicura perché sia `i` che `sizeof(a) / sizeof(a[0])` non hanno valori negativi. Per evitare il messaggio di warning possiamo aggiungere un cast che converte `sizeof(a) / sizeof(a[0])` in un intero `signed`:

```
for (i = 0; i < (int)(sizeof(a) / sizeof(a[0])); i++)
 a[i] = 0;
```

Scrivere `(int)(sizeof(a) / sizeof(a[0]))` è un po' scomodo, spesso è meglio definire una macro:

```
#define SIZE (int)(sizeof(a) / sizeof(a[0]))
for (i = 0; i < SIZE; i++)
 a[i] = 0;
```

Se dobbiamo comunque utilizzare una macro, qual è il vantaggio di usare `sizeof`? Risponderemo a questa domanda più avanti (il trucco consiste nell'aggiungere un parametro alla macro [macro parametrizzate > 14.3](#)).

PRATICAMMA

## Calcolare gli interessi

Il nostro prossimo programma stamperà una tabella che illustra il valore, su un certo periodo di anni, di un investimento di 100 dollari effettuato con diversi tassi di interesse. L'utente immetterà il tasso di interesse e il numero di anni nei quali i soldi verranno investiti. La tabella mostrerà il valore dell'investimento a intervalli di un anno (a quel tasso di interesse e per i quattro tassi di interesse successivi) assumendo che l'interesse venga composto una volta all'anno. Ecco come dovrebbe presentarsi una sessione del programma:

```
[nter interest rate: 6
[nter number of years: 5
```

| Years | 6%     | 7%     | 8%     | 9%     | 10%    |
|-------|--------|--------|--------|--------|--------|
| 1     | 106.00 | 107.00 | 108.00 | 109.00 | 110.00 |
| 2     | 112.36 | 114.49 | 116.64 | 118.81 | 121.00 |
| 3     | 119.10 | 122.50 | 125.97 | 129.50 | 133.10 |
| 4     | 126.25 | 131.08 | 136.05 | 141.16 | 146.41 |
| 5     | 133.82 | 140.26 | 146.93 | 153.86 | 161.05 |

Chiaramente possiamo usare l'istruzione `for` per stampare la prima riga. La seconda riga è un po' più complicata, dato che i suoi valori dipendono dai numeri della prima. La nostra soluzione è quella di memorizzare la prima riga in un vettore così come viene calcolata e poi usare i valori del vettore per calcolare la seconda. Naturalmente il processo può essere ripetuto per la terza riga e per quelle successive. Finiremo per avere due cicli `for`, uno annidato dentro l'altro. Il ciclo esterno conterrà da 1 fino al numero di anni richiesti dall'utente. Il ciclo interno incrementerà il tasso di interesse dal valore più piccolo a quello più grande.

```
interest.c /* Stampa una tavola di interessi composti */

#include <stdio.h>

#define NUM_RATES ((int) (sizeof(value) / sizeof(value[0])))
#define INITIAL_BALANCE 100.00

int main(void)
{
 int i, low_rate, num_years, year;
 double value[5];

 printf("Enter interest rate: ");
 scanf("%d", &low_rate);
 printf("Enter number of years: ");
 scanf("%d", &num_years);

 printf("\nYears");
 for (i = 0; i < NUM_RATES; i++) {
 printf("%6d%%", low_rate + i);
 value[i] = INITIAL_BALANCE;
 }
 printf("\n");

 for (year = 1; year <= num_years; year++) {
 printf("%3d ", year);
 for (i = 0; i < NUM_RATES; i++) {
 value[i] += (low_rate + i) / 100.0 * value[i];
 printf("%7.2f", value[i]);
 }
 printf("\n");
 }

 return 0;
}
```

Fate caso all'uso di `NUM_RATES` per controllare i due cicli. Se in un secondo momento volessimo cambiare la dimensione del vettore `value` i cicli si aggiusterebbero automaticamente.

## 8.2 Vettori multidimensionali

Un vettore può avere un qualsiasi numero di dimensioni. La seguente dichiarazione, per esempio, crea un vettore a due dimensioni (una *matrice* nella terminologia matematica):

```
int m[5][9];
```

Il vettore `m` ha 5 righe e 9 colonne. Sia le righe che le colonne vengono indicizzate a partire da 0, così come illustra la figura di pagina seguente.

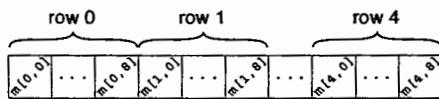
|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 |   |   |   |   |   |   |   |   |   |
| 1 |   |   |   |   |   |   |   |   |   |
| 2 |   |   |   |   |   |   |   |   |   |
| 3 |   |   |   |   |   |   |   |   |   |
| 4 |   |   |   |   |   |   |   |   |   |

Per accedere all'elemento di  $m$  che si trova alla riga  $i$  e alla colonna  $j$  dobbiamo scrivere  $m[i][j]$ . L'espressione  $m[i]$  indica la riga  $i$ -esima di  $m$ , mentre  $m[i][j]$  seleziona l'elemento  $j$  di quella riga.



Resistete alla tentazione di scrivere  $m[i, j]$  invece che  $m[i][j]$ . In questo contesto il C tratta la virgola come un operatore e quindi  $m[i, j]$  è equivalente a  $m[j]$  [operatori virgola > 6.3].

Sebbene visualizziamo i vettori a due dimensioni come delle tabelle, questo non è effettivamente il modo in cui vengono memorizzati all'interno del computer. Il C memorizza i vettori ordinandoli per righe, iniziando dalla riga 0, proseguendo con la riga 1 e così via. Ecco come viene memorizzato il vettore  $m$  dell'esempio:



Solitamente questo dettaglio viene ignorato ma alcune volte finisce per avere degli effetti sul codice.

Così come i cicli for si sposano perfettamente con i vettori a una dimensione, i cicli for annidati sono l'ideale per gestire i vettori a più dimensioni. Considerate per esempio il problema di inizializzare un vettore per usarlo come matrice identità (in matematica la *matrice identità* ha degli 1 nella diagonale principale dove gli indici di riga e colonna sono uguali, mentre è 0 altrove). Dobbiamo visitare in maniera sistematica ogni elemento del vettore. Una coppia di cicli for annidati (uno che si muove sulle righe e uno che si muove sulle colonne) è perfetta per questo compito:

```
#define N 10

double ident[N][N];
int row, col;

for (row = 0; row < N; row++)
 for (col = 0; col < N; col++)
 if (row == col)
 ident[row][col] = 1.0;
 else
 ident[row][col] = 0.0;
```

I vettori multidimensionali giocano un ruolo molto meno importante nel C rispetto a quello che accade in altri linguaggi di programmazione. Questo succede principalmente perché il C fornisce un modo molto più flessibile per memorizzare dei dati su più dimensioni: i vettori di puntatori [vettori di puntatori > 13.7].

## Inizializzare un vettore multidimensionale

Possiamo creare un inizializzatore per vettore a due dimensioni annidando degli inizializzatori unidimensionali.

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
 {0, 1, 0, 1, 0, 1, 0, 1, 0},
 {0, 1, 0, 1, 1, 0, 0, 1, 0},
 {1, 1, 0, 1, 0, 0, 0, 1, 0},
 {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

Ogni inizializzatore interno fornisce i valori per una riga della matrice. Gli inizializzatori per vettori con più di due dimensioni sono costruiti in modo del tutto simile.

Il C prevede una varietà di modi per abbreviare gli inizializzatori dei vettori multidimensionali.

- Se un inizializzatore non è grande abbastanza per riempire l'intero vettore multidimensionale, allora gli elementi rimanenti vengono impostati a 0. Per esempio, l'inizializzatore seguente riempie solo la prima delle tre righe del vettore `m`. Le due righe rimanenti conterranno degli zero:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
 {0, 1, 0, 1, 0, 1, 0, 1, 0},
 {0, 1, 0, 1, 1, 0, 0, 1, 0}};
```

- Se un lista interna non è sufficientemente lunga per riempire una riga, allora gli elementi rimanenti di quella riga vengono inizializzati a 0:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
 {0, 1, 0, 1, 0, 1, 0, 1},
 {0, 1, 0, 1, 1, 0, 0, 1},
 {1, 1, 0, 1, 0, 0, 0, 1},
 {1, 1, 0, 1, 0, 0, 1, 1}};
```

- Possiamo anche omettere le parentesi graffe interne:

```
int m[5][9] = {1, 1, 1, 1, 1, 0, 1, 1, 1,
 0, 1, 0, 1, 0, 1, 0, 1, 0,
 0, 1, 0, 1, 1, 0, 0, 1, 0,
 1, 1, 0, 1, 0, 0, 0, 1, 0,
 1, 1, 0, 1, 0, 0, 1, 1, 1};
```

Una volta che il compilatore ha visto un numero di elementi sufficiente da riempire una riga, inizia a riempire quella successiva.



Omettere le parentesi interne nell'inizializzatore di un vettore multidimensionale può essere rischioso visto che un elemento in più (o peggio ancora un elemento mancante) potrebbe compromettere la parte restante dell'inizializzatore. Con alcuni compilatori la mancanza delle parentesi produce un messaggio di warning come *missing braces around initializer*.



I designatori inizializzati del C99 funzionano anche con i vettori multidimensionali. Per esempio, per creare un matrice identità  $2 \times 2$  possiamo scrivere:

```
double ident[2][2] = {[0][0] = 1.0, [1][1] = 1.0};
```

Come al solito tutti gli elementi per i quali non viene specificato alcun valore vengono imposti a zero per default.

## Vettori costanti

Qualsiasi vettore, sia questo unidimensionale che multidimensionale, può essere reso costante iniziando la sua dichiarazione con la parola const:

```
const char hex_chars[] =
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
'A', 'B', 'C', 'D', 'E', 'F'};
```

Un vettore che è stato dichiarato costante non deve essere modificato dal programma, il compilatore rileva tutti i tentativi diretti di modificarne un elemento.

Dichiarare un vettore come costante presenta un paio di vantaggi. Per prima cosa documenta il fatto che il programma non modificherà il vettore, questo rappresenta un'importante informazione per chi dovesse leggere il codice in un secondo momento. Secondariamente aiuta il compilatore a individuare eventuali errori informandolo che non abbiamo intenzione di modificare il vettore. L'uso del qualificatore const non è limitato ai vettori. Come vedremo più avanti può essere applicato a qualsiasi variabile **[qualificatore const > 18.3]**. In ogni caso, const è particolarmente utile nelle dichiarazioni dei vettori perché questi possono contenere delle informazioni di riferimento che non devono cambiare durante l'esecuzione del programma.

HIGH-RAMMA

## Distribuire una mano di carte

Questo programma illustra sia i vettori a due dimensioni sia quelli costanti. Il programma distribuisce una mano di carte scelte a caso da un mazzo da gioco standard (nel caso in cui recentemente non avete avuto tempo per giocare, ogni carta di un mazzo standard ha un *seme* – cuori, quadri, fiori o picche – e un *valore* – due, tre, quattro, cinque, sei, sette, otto, nove, dieci, fante, regina, re oppure asso). L'utente specificherà di quante carte sarà composta la mano:

Enter number of cards in hand: 5

Your hand: 7c 2s 5d as 2h

Non è così immediato capire come il programma debba essere scritto. Come possiamo estrarre in modo casuale le carte? Come evitiamo di prendere due volte la stessa carta? Trattiamo questi problemi separatamente.

Per scegliere a caso le carte useremo diverse funzioni di libreria: la funzione time (da `<time.h>`) che restituisce l'ora corrente codificata come un singolo numero **[funzione**

**time > 26.3**; la funzione `srand` (da `<stdlib.h>`) che inizializza il generatore random del C [funzione `srand` > 26.2]. Passando il valore ritornato da `time` alla funzione `srand` garantisce di non consegnare le stesse carte ogni volta che eseguiamo il programma. La funzione `rand` (anch'essa da `<stdlib.h>`) produce un numero apparentemente casuale ogni volta che viene invocata [funzione `rand` > 26.2]. Usando l'operatore `%` possiamo scalare il valore restituito della `rand` in modo che cada tra 0 e 3 (per i semi) o tra 0 e 12 (per i valori).

Per evitare di scegliere due volte la stessa carta terremo traccia di quelle che sono già state pescate. A questo scopo useremo un vettore chiamato `in_hand` con quattro righe (una per ogni seme) e 13 colonne (una per ogni valore). In altre parole, ogni elemento del vettore corrisponde a una delle 52 carte del mazzo. Tutti gli elementi del vettore verranno impostati al valore `false` all'inizio del programma. Ogni volta che peschiamo a caso una carta, controlliamo se l'elemento corrispondente nel vettore `in_hand` è vero o falso. Se è vero, allora dovremo pescare un'altra carta. Se è falso, memorizzeremo il valore `true` all'interno dell'elemento del vettore. In questo modo potremo ricordarci che la carta è già stata scelta.

Una volta verificato che la carta è "nuova" (non ancora selezionata) abbiamo bisogno di tradurre il suo valore e il suo valore numerico nei caratteri corrispondenti in modo da poterla stampare. Per tradurre il valore della carta e il suo seme in questo nuovo formato, creeremo due vettori di caratteri (uno per il valore e uno per il seme) e vi accederemo usando i valori numerici come indici per i vettori appena menzionati. Questi vettori non cambieranno durante l'esecuzione del programma e perciò possiamo dichiararli costanti:

```
deal.c /* Distribuisce una mano di carte scelta casualmente */

#include <stdbool.h> /* solo C99 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_SUITS 4
#define NUM_RANKS 13

int main(void)
{
 bool in_hand[NUM_SUITS][NUM_RANKS] = {false};
 int num_cards, rank, suit;
 const char rank_code[] = {'2','3','4','5','6','7','8',
 '9','t','j','q','k','a'};
 const char suit_code[] = {'c','d','h','s'};
 srand((unsigned) time(NULL));

 printf("Enter number of cards in hand: ");
 scanf("%d", &num_cards);

 printf("Your hand:");
 while (num_cards > 0) {
 suit = rand() % NUM_SUITS; /* sceglie un seme random */
 rank = rand() % NUM_RANKS; /* sceglie un valore random */
 if (!in_hand[suit][rank]) {
 in_hand[suit][rank] = true;
 printf("%c%c ", suit_code[suit], rank_code[rank]);
 num_cards--;
 }
 }
}
```

```

 rank = rand() % NUM_RANKS; /* sceglie un valore random */
 if (!in_hand[suit][rank]) {
 in_hand[suit][rank] = true;
 num_cards--;
 printf(" %c%c", rank_code[rank], suit_code[suit]);
 }
}
printf("\n");
return 0;
}

```

Fate caso all'inizializzatore per il vettore `in_hand`:

```
bool in_hand[NUM_SUITS][NUM_RANKS] = {false};
```

Anche se `in_hand` è un vettore a due dimensioni possiamo usare una singola coppia di parentesi graffe (al rischio di un possibile messaggio di warning da parte del compilatore). Inoltre abbiamo inserito solo un valore nell'inizializzatore, sapendo che il compilatore riempirà il resto del vettore con degli zeri (equivalenti a `false`).

### 8.3 Vettori a lunghezza variabile (C99)

Nella Sezione 8.1 abbiamo detto che la lunghezza di un vettore deve essere specificata da un'espressione costante. Tuttavia nel C99 a volte è possibile usare un'espressione che *non* è costante. La seguente rivisitazione del programma `reverse.c` (Sezione 8.1) illustra questa possibilità:

```

reverse2.c /* Inverte l'ordine di una sequenza di numeri usando un vettore
 a lunghezza variabile - solo C99 */

#include <stdio.h>

int main(void)
{
 int i, n;

 printf("How many numbers do you want to reverse? ");
 scanf("%d", &n);

 int a[n]; /* C99 only - length of array depends on n */

 printf("Enter %d numbers: ", n);
 for (i = 0; i < n; i++)
 scanf("%d", &a[i]);

 printf("In reverse order:");
 for (i = n - 1; i >= 0; i--)
 printf(" %d", a[i]);
 printf("\n");

 return 0;
}

```

Il vettore `a` di questo programma è un esempio di **vettore a lunghezza variabile (VLA, da variable-length array)**. La lunghezza di un VLA viene calcolata quando il programma è in esecuzione e non quando viene compilato. Il vantaggio principale di un VLA consiste nel fatto che il programmatore non deve scegliere una lunghezza arbitraria quando dichiara il vettore, infatti è il programma stesso a calcolare esattamente quanti elementi sono necessari. Se è il programmatore a fare la scelta c'è una buona probabilità che il vettore sia troppo lungo (sprecando memoria) o troppo corto (causando il malfunzionamento del programma). Nel programma `reverse2.c` è il numero immesso dall'utente a determinare la lunghezza di `a`, il programmatore non è costretto a scegliere una lunghezza fissa come nella versione originale del programma.

La lunghezza di un VLA non deve essere specificata da una singola variabile. Infatti sono ammesse anche espressioni arbitrarie contenenti anche operatori. Per esempio:

```
int a[3*i+5];
int b[j+k];
```

Come gli altri vettori anche i VLA possono essere multidimensionali:

```
int c[m][n];
```

La restrizione principale imposta ai VLA è che questi non possono avere una durata di memorizzazione statica (non abbiamo ancora incontrato vettori con questa proprietà) [durata di memorizzazione statica > 18.2]. Un'altra restrizione per i VLA riguarda l'impossibilità di avere un inizializzatore.

I vettori a lunghezza variabile vengono visti spesso nelle funzioni diverse dal `main`. Un notevole vantaggio di un VLA che appartiene a una funzione `f` è che può avere una dimensione diversa ogni volta che `f` viene invocata. Esploreremo questa caratteristica nella Sezione 9.3.

## Domande & Risposte

**D: Perché gli indici dei vettori partono da 0 e non da 1? [p. 168]**

R: Far iniziare gli indici da 0 semplifica un po' il compilatore e inoltre rende l'individuazione leggermente più veloce.

**D: E se volessimo un vettore i cui indici vanno da 1 a 10 invece che da 0 a 9?**

R: C'è un trucco comune: dichiarare un vettore con 11 elementi invece di 10. Gli indici andranno così da 0 a 10, di conseguenza è sufficiente ignorare l'elemento 0.

**D: È possibile usare un carattere come indice di un vettore?**

R: Sì perché il C tratta i caratteri come interi. Probabilmente però avrete bisogno di "scalare" il carattere prima di poterlo usare come indice. Diciamo, per esempio, di voler usare il vettore `letter_count` per mantenere un contatore per ogni lettera dell'alfabeto. Il vettore avrà bisogno di 26 elementi e così lo dichiareremo in questo modo:

```
int letter_count[26];
```

Non possiamo però usare direttamente le lettere come indici del vettore `letter_count` a causa del fatto che non rientrano nell'intervallo compreso tra 0 e 25. Per scalare una lettera minuscola al range appropriato è sufficiente sottrarre il carattere

'a'. Invece, per scalare una lettera maiuscola sottrarremo il carattere 'A'. Per esempio, se la variabile ch contiene una lettera minuscola allora per azzerare il valore corrispondente scriviamo:

```
letter_count[ch - 'a'] = 0;
```

Un inconveniente minore è costituito dal fatto che questa tecnica non è completamente portabile perché presume che le lettere abbiano codici consecutivi. In ogni caso funziona con molti set di caratteri, incluso il set ASCII.

**D: Sembra che un designatore inizializzato possa inizializzare più di una volta l'elemento di un vettore. Considerate la seguente dichiarazione:**

```
int a[] = {4, 9, 1, 8, [0] = 5, 7};
```

**questa dichiarazione è ammessa? In tal caso che lunghezza avrà il vettore? (p. 172)**

**R:** Sì, questa dichiarazione è ammissibile. Ecco come funziona: durante l'elaborazione dell'inizializzatore, il compilatore tiene traccia del successivo elemento da inizializzare. Normalmente l'elemento che deve essere inizializzato è successivo a quello appena gestito. Tuttavia, quando nella lista appare un designatore questo forza l'elemento successivo a essere specificato dall'indice e questo succede *anche se quell'elemento è già stato inizializzato*.

Ecco di seguito il comportamento tenuto (passo dopo passo) dal compilatore durante l'elaborazione dell'inizializzatore per il vettore a:

L'elemento 0 viene inizializzato a 4: il successivo da inizializzare è l'elemento 1.

L'elemento 1 viene inizializzato a 9: il successivo da inizializzare è l'elemento 2.

L'elemento 2 viene inizializzato a 1: il successivo da inizializzare è l'elemento 3.

L'elemento 3 viene inizializzato a 8: il successivo da inizializzare è l'elemento 4.

Il designatore [0] fa sì che il successivo da inizializzare sia l'elemento 0 e quindi l'elemento 0 viene inizializzato a 5 (rimpiazzando così il 4 che era stato memorizzato precedentemente). L'elemento 1 dovrà essere il successivo a essere inizializzato.

L'elemento 1 viene inizializzato a 7 (rimpiazzando il 9 che era stato memorizzato precedentemente). Il successivo a essere inizializzato è l'elemento 2 (ma questo è irrilevante visto che siamo alla fine della lista).

L'effetto netto è equivalente ad aver scritto

```
int a[] = {5, 7, 1, 8};
```

Quindi la lunghezza del vettore è uguale a quattro.

**D: Se proviamo a copiare un vettore in un altro con l'operatore di assegnazione, il compilatore emette un messaggio di errore. Cosa c'è di sbagliato?**

**R:** Sebbene sembri assolutamente plausibile, l'assegnazione

```
a = b; /* a e b sono vettori */
```

in realtà non è ammessa. La ragione non è ovvia e ha a che vedere con la relazione che nel C intercorre tra i vettori e i puntatori. Questo è un argomento che esplorremo nel Capitolo 12.

Il modo più semplice per copiare un vettore in un altro è quello di usare un ciclo che effettui la copia elemento per elemento:

```
for (i=0; i < N; i++)
a[i] = b[i];
```

Un'altra possibilità è quella di usare la funzione `memcpy` (*memory copy*) presente nell'header `<string.h>` [funzione `memcpy` > 23.6]. La `memcpy` è una funzione a basso livello che semplicemente copia dei byte da un posto a un altro. Per copiare il vettore `b` nel vettore `a` possiamo usare la `memcpy` come segue:

```
memcpy(a, b, sizeof(a));
```

Molti programmati preferiscono usare `memcpy`, specialmente per i vettori di grandi dimensioni, perché è potenzialmente più veloce di un normale ciclo.

**\*D: La Sezione 6.4 ha menzionato il fatto che il C99 non ammette l'uso dell'istruzione goto per bypassare la dichiarazione di un vettore a lunghezza variabile. Qual è la ragione per questa restrizione?**

**R:** La memoria utilizzata per un vettore a lunghezza variabile di solito viene allocata nel momento in cui l'esecuzione del programma raggiunge la dichiarazione del vettore. Bypassare la dichiarazione usando un'istruzione goto potrebbe comportare l'accesso, da parte del programma, a elementi mai allocati.

## Esercizi

### Sezione 8.1



1. Abbiamo discusso l'uso dell'espressione `sizeof(a) / sizeof(a[0])` per calcolare il numero di elementi di un vettore. Funzionerebbe anche l'espressione `sizeof(a) / sizeof(t)`, dove `t` è il tipo degli elementi di `a`, tuttavia questa è considerata un'operazione di qualità inferiore. Perché?
2. La Sezione D&R mostra come usare una *lettera* come indice di un vettore. Descrivete come utilizzare una *cifra* (presa sotto forma di carattere) come indice.
3. Scrivete la dichiarazione di un vettore chiamato `weekend` contenente sette valori di tipo `bool`. Includete anche un inizializzatore che imposti il primo e l'ultimo elemento al valore `true`, mentre gli altri elementi dovranno essere `false`.
4. (C99) Ripetete l'Esercizio 3, questa volta utilizzando un designatore inizializzato. Rendete l'inizializzatore il più breve possibile.
5. La serie di Fibonacci è  $0, 1, 1, 2, 3, 5, 8, 13, \dots$  dove ogni numero è pari alla somma dei due numeri precedenti. Scrivete un frammento di programma che dichiari il vettore `fib_numbers` lungo 40 elementi e lo riempia con i primi 40 numeri della serie. Suggerimento: riempite i primi due numeri individualmente e poi usate un ciclo per calcolare i rimanenti.

### Sezione 8.2

6. Calcolatrici, orologi e altri dispositivi elettronici utilizzano spesso display a sette segmenti per l'output numerico. Per formare una cifra questi dispositivi accendono solamente alcuni dei sette segmenti lasciando spenti gli altri:

0 1 2 3 4 5 6 7 8 9

Supponete di dover creare un vettore che ricordi quali elementi debbano essere accesi per formare ogni cifra. Numeriamo i segmenti come segue:

|   |
|---|
| 0 |
| 6 |
| 1 |
| 4 |
| 3 |
| 2 |

Ecco come potrebbe apparire un vettore nel quale ogni riga rappresenta una cifra:

```
const int segments[10][7] = {{1, 1, 1, 1, 1, 1, 0}, ...};
```

Questa era la prima riga dell'inizializzatore, completate inserendo quelle che mancano.

- W 7. Usando le scorciatoie descritte nella Sezione 8.2 restringere il più possibile l'inizializzatore del vettore segments (Esercizio 6).
- 8. Scrivete la dichiarazione di un vettore a due dimensioni chiamato temperature\_readings in modo che memorizzi un mese di letture orarie di temperatura (per semplicità assumete che un mese abbia 30 giorni). Le righe del vettore dovrebbero rappresentare il giorno del mese mentre le colonne dovrebbero rappresentare le ore del giorno.
- 9. Utilizzando il vettore dell'Esercizio 8 scrivete un frammento di programma che calcoli la temperatura media di un mese (media fatta su tutti i giorni del mese e tutte le ore del giorno).
- 10. Scrivete la dichiarazione di un vettore di char 8×8 chiamato chess\_board. Includete un inizializzatore che metta i seguenti dati all'interno del vettore (un carattere per ogni elemento del vettore):

```
x n b q k b n r
p p p p p p p p
.
.
.
.
.
.
p p p p p p p p
r n b q k b n r
```

- 11. Scrivete la dichiarazione di un vettore di char 8×8 chiamato checker\_board e poi utilizzate un ciclo per memorizzare i seguenti dati all'interno del vettore (un carattere per ogni elemento del vettore):

```
b r b r b r b r
r b r b r b r b
b r b r b r b r
r b r b r b r b
b r b r b r b r
r b r b r b r b
b r b r b r b r
r b r b r b r b
```

*Suggerimento:* l'elemento in riga  $i$  e colonna  $j$  deve essere uguale alla lettera  $B$  se  $i + j$  è un numero pari.

## Progetti di programmazione

1. Modificate il programma repdigit.c della Sezione 8.1 in modo che stampi le cifre che sono ripetute (se ce ne sono):

Enter a number: 939577

Repeated digit(s): 7 9

2. Modificate il programma repdigit.c della Sezione 8.1 in modo che stampi una tabella che illustra per ogni cifra quante volte appare all'interno del numero:

Enter a number: 41271092

Digit: 0 1 2 3 4 5 6 7 8 9

Occurrences: 1 2 2 0 1 0 0 1 0 1

3. Modificate il programma repdigit.c della Sezione 8.1 in modo che l'utente possa immettere più di un numero da analizzare per le cifre ripetute. Il programma deve terminare quando l'utente immette un numero minore o uguale a 0.
4. Modificate il programma reverse.c della Sezione 8.1 per usare l'espressione `(int)(sizeof(a) / sizeof(a[0]))` (o una macro con questo valore) per ottenere la lunghezza del vettore.
5. Modificate il programma interests.c della Sezione 8.1 in modo da calcolare gli interessi composti *mensili* invece che *annuali*. Il formato dell'output non deve cambiare, il bilancio deve essere visibile ancora in intervalli annuali.
6. Lo stereotipo di un novellino di Internet è un tizio chiamato B1FF, il quale ha un unico modo per scrivere i messaggi. Ecco un tipico comunicato di B1FF:

H3Y DUD3, C 15 R1LLY COOL!!!!!!!

Scrivete un "filtro B1FF" che legga un messaggio immesso dall'utente e lo traduca nel modo di scrivere di B1FF:

Enter a message: Hey dude, C is rilly cool

In B1FF-speak: H3Y DUD3, C 15 R1LLY COOL!!!!!!!

Il programma deve convertire il messaggio in lettere maiuscole e sostituire certe lettere con delle cifre ( $A \rightarrow 4, B \rightarrow 8, E \rightarrow 3, I \rightarrow 1, O \rightarrow 0, S \rightarrow 5$ ). Alla fine del messaggio devono essere inseriti 10 punti esclamativi. *Suggerimento:* memorizzate il messaggio originale in un vettore di caratteri e poi ripassate il vettore stampando i caratteri uno alla volta.

7. Scrivete un programma che legga un vettore di interi  $5 \times 5$  e poi stampi la somma delle righe e delle colonne:

Enter row 1: 8 3 9 0 10

Enter row 2: 3 5 17 1 1

Enter row 3: 2 8 6 23 1

Enter row 4: 15 7 3 2 9

Enter row 5: 6 14 2 6 0

Row totals: 30 27 40 36 28

Column totals: 34 37 37 32 21

- 8. Modificate il Progetto di programmazione 7 in modo che stampi il punteggio ottenuto da cinque studenti in cinque quiz. Il programma successivamente deve calcolare il punteggio totale e quello medio per ogni *studente*. Inoltre andranno calcolati il punteggio medio, quello massimo e quello minimo per ogni *quiz*.
- 9. Scrivete un programma che generi un "cammino casuale" in un vettore  $10 \times 10$ . Il vettore conterrà dei caratteri (inizialmente saranno tutti '.'). Il programma deve passare casualmente da un elemento all'altro, muovendosi in alto, in basso, a sinistra o a destra di una posizione soltanto. Gli elementi visitati dal programma dovranno essere etichettati con le lettere che vanno dalla *A* alla *Z* nell'ordine con cui vengono visitati. Ecco un esempio dell'output desiderato:

```

A
B C D
. F E
H G
I
J Z .
K . . R S T U V Y .
L M P Q . . . W X .
. N O
.

```

*Suggerimento:* per generare i numeri casuali usate le funzioni `srand` e `rand` (guardate `deal.c`). Dopo aver generato un numero prendete il resto ottenuto e dividetelo per 4. I quattro possibili valori per il resto (0, 1, 2 e 3) indicano la direzione della prossima mossa. Prima di effettuare la mossa controllate che (a) non vada fuori dal vettore e (b) non ci porti in un elemento al quale è stata già assegnata una lettera. Se una delle due condizioni viene violata provate allora a muovervi in un'altra direzione. Se tutte e quattro le direzioni sono bloccate il programma deve terminare. Ecco un esempio di fine prematura:

```

A B G H I . . .
. C F . J K . .
. D E . M L . .
. . . N O . .
. . W X Y P Q . .
. . V U T S R . .
.
.
.
.

```

Y è bloccata da tutti a quattro i lati e quindi non c'è modo di inserire la Z.

10. Modificate il Progetto di programmazione 8 del Capitolo 5 in modo che gli orari di partenza vengano memorizzati in un vettore e gli orari di arrivo vengano memorizzati in un secondo vettore (gli orari sono degli interi che rappresentano il numero di minuti dalla mezzanotte). Il programma dovrà usare un ciclo per cercare nel vettore degli orari di partenza quello che è più vicino all'orario immesso dall'utente.

11. Modificate il Progetto di Programmazione 4 del Capitolo 7 in modo che il programma etichetti il suo output:

Enter phone number: 1-800-CÖL-LECT

In numeric form: 1-800-265-5328

Il programma avrà bisogno di memorizzare il numero di telefono (sia nella forma originale che in quella numerica) in un vettore di caratteri fino a quando non può essere stampato. Potete assumere che il numero di telefono non sia più lungo di 15 caratteri.

12. Modificate il Progetto di Programmazione 5 del Capitolo 7 in modo che i valori delle lettere dello Scarabeo vengano memorizzati in un vettore. Tale vettore avrà 26 elementi corrispondenti alle 26 lettere dell'alfabeto. Per esempio l'elemento 0 del vettore conterrà un 1 (perché il valore della lettera A è 1), l'elemento 1 conterrà un 3 (perché il valore della lettera B è 3) e così via. Quando viene letto un carattere della parola in input il programma dovrà usare il vettore per determinarne il valore. Usate un inizializzatore per costruire il vettore.

13. Modificate il Progetto di Programmazione 11 del Capitolo 7 in modo che il programma etichetti il suo output:

Enter a first and last name: Lloyd Fosdick

You entered the name: Fosdick, L.

Il programma avrà bisogno di memorizzare il cognome (ma non il nome) in un vettore di caratteri fino al momento in cui non verrà stampato. Potete assumere che il cognome non sia più lungo di 20 caratteri.

14. Scrivete un programma che inverta le parole presenti in una frase:

Enter a sentence: you can cage a swallow can't you?

Reversal of sentence: you can't swallow a cage can you?

*Suggerimento:* usate un ciclo per leggere i caratteri uno alla volta e per memorizzarli in un vettore char unidimensionale. Interrompete il ciclo quando incontrate un punto, un punto di domanda, un punto esclamativo (il "carattere di terminazione") il quale deve essere salvato in una variabile separata di tipo char. Successivamente usate un secondo ciclo per percorrere all'indietro il vettore dall'ultima parola alla prima. Stampate l'ultima parola e poi andate in cerca della penultima. Ripetete fino a quando non viene raggiunto l'inizio del vettore. Come ultima cosa stampate il carattere terminatore.

15. Il cifrario di Cesare, attribuito a Giulio Cesare, è una delle più antiche tecniche crittografiche e si basa sulla sostituzione di ogni lettera del messaggio con un'altra

lettera che si trova più avanti nell'alfabeto di un numero prefissato di posizioni (se scorrendo una lettera si andasse oltre la *Z*, sappiate che il cifrario si "arrotola" ricominciando dall'inizio dell'alfabeto. Per esempio se ogni lettera viene sostituita da quella che si trova due posizioni più avanti, allora la *Y* verrebbe sostituita da una *A*, mentre la *Z* verrebbe sostituita dalla *B*). Scrivete un programma che cripti un messaggio usando il cifrario di Cesare. L'utente immetterà il messaggio che deve essere cifrato e lo scorrimento (il numero di posizioni delle quali le lettere devono essere fatte scorrere):

Enter message to be encrypted: Go ahead, make my day.

Enter shift amount (1-25): 3

Encrypted message: J<sub>r</sub> dkhdg, pdnh pb gdb.

Osservate che il programma può decifrare il messaggio se l'utente immette un valore pari a 26 meno la chiave originale:

Enter message to be encrypted: J<sub>r</sub> dkhdg, pdnh pb gdb.

Enter shift amount (1-25): 23

Encrypted message: Go ahead, make my day.

Potete assumere che il messaggio non ecceda gli 80 caratteri. I caratteri che non sono lettere devono essere lasciati così come sono. Le lettere minuscole rimangono minuscole anche quando vengono cifrate, analogamente le lettere maiuscole rimangono maiuscole. Suggerimento: per gestire il problema dell'arrotolamento usate l'espressione  $((ch - 'A') + n) \% 26 + 'A'$  per calcolare la versione cifrata delle lettere maiuscole, dove *ch* è la lettera da cifrare e *n* è lo scorrimento (avrete bisogno di un'espressione simile per le lettere minuscole).

16. Scrivete un programma che controlli se due parole sono degli anagrammi (cioè delle permutazioni delle stesse lettere):

Enter first word: smartest

Enter second word: mattress

The words are anagrams.

Enter first word: dumbest

Enter second word: stumble

The words are not anagrams.

Scrivete un ciclo che legga la prima parola carattere per carattere, usando un vettore di 26 interi per tenere traccia del numero di occorrenze di ogni lettera. (Per esempio dopo che la parola *smartest* è stata letta il vettore dovrebbe contenere i valori 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 2 2 0 0 0 0 riflettendo il fatto che *smartest* contiene una *a*, una *e*, una *m*, una *r*, due *s* e due *t*). Usate un altro ciclo per leggere la seconda parola, ma questa volta per ogni lettera letta decrementate l'elemento corrispondente nel vettore. Entrambi i cicli devono ignorare i caratteri che non sono lettere ed entrambi devono trattare le lettere maiuscole allo stesso modo in cui trattano le minuscole. Dopo che la seconda parola è stata letta usate un terzo ciclo per controllare se tutti gli elementi del vettore sono uguali a zero. Se è così allora le due parole sono anagrammi. Suggerimento: se volete potete usare le funzioni presenti nell'header `<cctype.h>` come `isalpha` o `tolower`.

17. Scrivete un programma che stampi un quadrato magico  $n \times n$  (un quadrato di numeri  $1, 2, \dots, n^2$  dove la somma delle righe, delle colonne e delle diagonali è sempre la stessa). L'utente specificherà il valore di  $n$ :

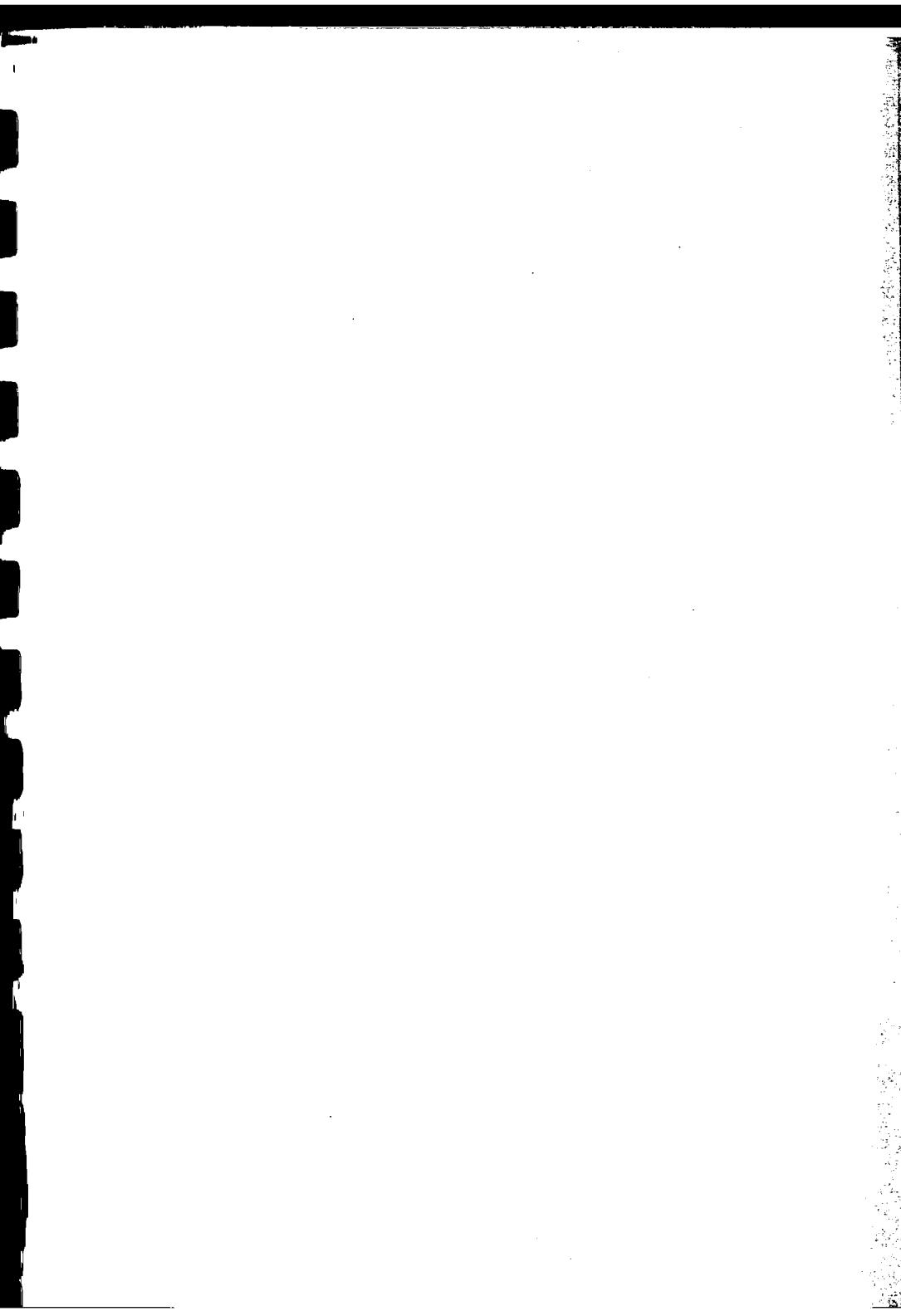
This program creates a magic square of a specified size.

The size must be an odd number between 1 and 99.

Enter the size of magic square: 5

|    |    |    |    |    |
|----|----|----|----|----|
| 17 | 24 | 1  | 8  | 15 |
| 23 | 5  | 7  | 14 | 16 |
| 4  | 6  | 13 | 20 | 22 |
| 10 | 12 | 19 | 21 | 3  |
| 11 | 18 | 25 | 2  | 9  |

Memorizzate il quadrato magico in un vettore a due dimensioni. Iniziate mettendo il numero 1 in mezzo alla riga 0. Disponete i numeri rimanenti  $2, 3, \dots, n^2$  muovendovi in su di una riga e di una colonna. Qualsiasi tentativo di andare fuori dai limiti del vettore deve "arrotolarsi" sulla faccia opposta del vettore stesso. Per esempio, invece di memorizzare il prossimo numero nella riga  $-1$ , dovremmo salvarlo nella riga  $n - 1$  (l'ultima riga). Invece di memorizzare il prossimo numero nella colonna  $n$ , lo memorizzeremo nella colonna 0. Se un particolare elemento del vettore è già occupato mettete il numero immediatamente sotto il numero che è stato memorizzato precedentemente. Se il vostro compilatore supporta i vettori a lunghezza variabile, dichiarate un vettore con  $n$  righe ed  $n$  colonne. In caso contrario dichiarate il vettore in modo che abbia 99 righe e 99 colonne.



# 9 Funzioni

Nel Capitolo 2 abbiamo visto che una funzione non è altro che un raggruppamento di una serie di istruzioni al quale è stato assegnato un nome. Sebbene il termine “funzione” derivi dalla matematica, le funzioni C non sempre somigliano a funzioni matematiche. Nel C una funzione non deve necessariamente avere degli argomenti e nemmeno deve restituire un valore (in alcuni linguaggi di programmazione una “funzione” calcola un valore mentre una “procedura” non lo fa. Nel C manca questa distinzione).

Le funzioni sono i blocchi che costituiscono i programmi C. Ogni funzione è essenzialmente un piccolo programma con le sue dichiarazioni e le sue istruzioni. Usando le funzioni possiamo suddividere un programma in pezzi più piccoli che sono più facili da scrivere e modificare (sia per noi che per gli altri). Le funzioni ci permettono di evitare la duplicazione del codice che viene usato più di una volta. Le funzioni, inoltre, sono riutilizzabili: in un programma possiamo usare una funzione che originariamente faceva parte di un programma diverso.

I nostri programmi finora erano costituiti dalla sola funzione `main`. In questo capitolo vedremo come scrivere funzioni diverse dal `main` e impareremo nuovi concetti a riguardo del `main` stesso. La Sezione 9.1 illustra come definire e chiamare delle funzioni. La Sezione 9.2 tratta le dichiarazioni delle funzioni e di come queste differiscano dalle definizioni delle funzioni. Nella Sezione 9.3 esamineremo come gli argomenti vengono passati alle funzioni. La parte rimanente del capitolo tratta l’istruzione `return` (Sezione 9.4), gli argomenti collegati alla terminazione del programma (Sezione 9.5) e la ricorsione (Sezione 9.6).

## 9.1 Definire e invocare le funzioni

Prima di addentrarci nelle regole formali per la definizione delle funzioni, guardiamo tre semplici programmi che definiscono delle funzioni.

## PROGRAMMA

**Calcolo delle medie**

Supponete di dover calcolare spesso la media tra due valori double. La libreria del C non ha una funzione "media" (*average*), ma possiamo crearne facilmente una. Ecco come potrebbe apparire:

```
double average(double a, double b)
{
 return (a + b) / 2;
}
```

La parola **double** presente all'inizio è il **tipo restituito** dalla funzione **average**: ovvero il tipo dei dati che vengono restituiti dalla funzione ogni volta che viene invocata.

**D&R** Gli identificatori **a** e **b** (i **parametri** della funzione) rappresentano due numeri che dovranno essere forniti alla funzione quando viene chiamata. Ogni parametro deve possedere un tipo (esattamente come ogni variabile). In questo esempio sia **a** che **b** sono di tipo **double** (può sembrare strano ma la parola **double** deve apparire due volte: una per **a** e una per **b**). Un parametro di una funzione è essenzialmente una variabile il cui valore iniziale verrà fornito successivamente quando la funzione viene invocata.

Ogni funzione possiede una parte eseguibile chiamata **corpo** (o **body**), la quale viene racchiusa tra parentesi graffe. Il corpo di **average** consiste della sola istruzione **return**. Eseguire questa istruzione impone alla funzione di "ritornare" al punto in cui è stata invocata. Il valore di  $(a + b) / 2$  sarà il valore restituito dalla funzione.

Per chiamare una funzione scriviamo il suo nome seguito da un elenco di **argomenti**. Per esempio, **average(x, y)** è una chiamata alla funzione **average**. Gli argomenti vengono usati per fornire informazioni alla funzione. Nel nostro caso la funzione **average** ha bisogno di sapere quali sono i due numeri dei quali si deve calcolare la media. L'effetto della chiamata **average(x, y)** è quello di creare una copia dei valori di **x** e **y** dentro i parametri **a** e **b** e, successivamente, eseguire il corpo della funzione. Un argomento non deve necessariamente essere una variabile, una qualsiasi espressione compatibile andrà bene, infatti possiamo scrivere sia **average(5.1, 8.9)** che **average(x/2, y/3)**.

Potremo inserire una chiamata ad **average** ovunque sia necessario. Per esempio potremmo scrivere

```
printf("Average: %g\n", average(x, y));
```

per calcolare e stampare la media di **x** e **y**. Questa istruzione ha il seguente effetto:

1. la funzione **average** viene chiamata prendendo **x** e **y** come argomenti;
2. **x** e **y** vengono copiati dentro **a** e **b**;
3. la funzione **average** esegue la sua istruzione **return**, restituendo la media di **a** e **b**;
4. la **printf** stampa il valore ritornato da **average** (il valore restituito diventa uno degli argomenti della **printf**).

Osservate che il valore restituito da **average** non viene salvato, il programma lo stampa e successivamente lo scarta. Se avessimo avuto bisogno di quel valore in un punto successivo del programma, avremmo potuto salvarlo all'interno di una variabile:

```
avg = average(x, y);
```

Questa istruzione chiama `average` e salva il valore restituito nella variabile `avg`.

Ora utilizzeremo la funzione `average` in un programma completo. Il programma seguente legge tre numeri e calcola la loro media effettuandola sulle diverse coppie:

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

Tra le altre cose, questo programma dimostra che una funzione può essere invocata tutte le volte di cui ne abbiamo bisogno.

```
average.c /* Calcola la media delle coppie formate a partire da tre numeri */
#include <stdio.h>

double average(double a, double b)
{
 return (a + b) / 2;
}

int main(void)
{
 double x, y, z;

 printf("Enter three numbers: ");
 scanf("%lf%lf%lf", &x, &y, &z);
 printf("Average of %g and %g: %g\n", x, y, average(x, y));
 printf("Average of %g and %g: %g\n", y, z, average(y, z));
 printf("Average of %g and %g: %g\n", x, z, average(x, z));

 return 0;
}
```

Osservate che abbiamo messo la definizione di `average` prima del `main`. Vedremo nella Sezione 9.2 che mettere `average` dopo la funzione `main` causerebbe dei problemi.

## PROGRAMMA

### Stampare un conto alla rovescia

Non tutte le funzioni restituiscono un valore. Per esempio, una funzione il cui scopo sia quello di produrre dell'output non avrebbe bisogno di restituire alcunché. Per indicare che una funzione non ha valore da restituire, specifichiamo che il suo tipo restituito è `void` (`void` è un tipo privo di valori). Considerate la seguente funzione che stampa il messaggio `T minus n and counting`, dove `n` viene fornito quando viene chiamata la funzione:

```
void print_count(int n)
{
 printf("T minus %d and counting\n", n);
}
```

La funzione `print_count` ha un solo parametro, `n`, di tipo `int`. Non restituisce nulla e per questo abbiamo usato `void` come tipo restituito e abbiamo omesso l'istruzione

`return.` Dato che non restituisce nessun valore non possiamo chiamare la `print_count` nello stesso modo in cui chiamavamo `average`. Una chiamata a `print_count` deve apparire come un'istruzione a sé stante:

```
print_count(i);
```

Ecco un programma che chiama `print_count` per 10 volte all'interno di un ciclo:

```
/* Stampa un conto alla rovescia */

#include <stdio.h>

void print_count(int n)
{
 printf("T minus %d and counting\n", n);
}

int main(void)
{
 int i;
 for (i = 10; i > 0; --i)
 print_count(i);

 return 0;
}
```

Inizialmente `i` ha una valore pari a 10. Quando la `print_count` viene chiamata per la prima volta, la variabile `i` viene copiata in `n` e questo fa sì che anche la variabile `n` abbia il valore 10. Ne risulta che la prima chiamata alla `print_count` stamperà

`T minus 10 and counting`

Successivamente `print_count` ritorna al punto in cui è stata invocata, ovvero nel corpo del ciclo `for`. L'istruzione `for` riprende da dove era stata interrotta decrementando la variabile `i` al valore 9 e controllando, successivamente, se la variabile è maggiore di 0. Lo è e quindi la `print_count` viene chiamata nuovamente stampando questa volta il messaggio

`T minus 9 and counting`

Ogni volta che la `print_count` viene chiamata, la variabile `i` possiede un valore diverso e quindi la funzione `print_count` stamperà 10 messaggi differenti.

PIRELLAMMA

## Stampare un motto (rivisitato)

Alcune funzioni non hanno alcun parametro. Considerate la funzione `print_pun` che stampa il motto scherzoso conosciuto come *bad pun* ogni volta che viene invocata:

```
void print_pun(void)
{
 printf("To C, or not to C: that is the question.\n");
}
```

La parola void all'interno delle parentesi indica che print\_pun non ha nessun argomento (questa volta stiamo usando void come un segnaposto che significa "qui non ci va nulla").

Per chiamare una funzione senza argomenti dobbiamo scrivere il nome della funzione seguita dalle parentesi vuote:

```
print_pun();
```

Le parentesi *devono* essere presenti anche se non ci sono argomenti.

Ecco un piccolo programma test per la funzione print\_pun:

```
pun2.c /* Stampa il bad pun */

#include <stdio.h>

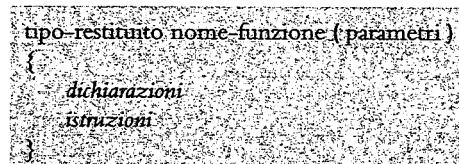
void print_pun(void)
{
 printf("To C, or not to C: that is the question.\n");
}

int main(void)
{
 print_pun();
 return 0;
}
```

L'esecuzione di questo programma inizia con la prima istruzione del main che è proprio una chiamata alla print\_pun. Quando print\_pun inizia l'esecuzione, chiama a sua volta la printf per stampare una stringa. Quando la printf ritorna, ritorna anche la print\_pun.

## Definizione di funzioni

Ora che abbiamo visto alcuni esempi, guardiamo alla forma generale di **definizione di una funzione**:



Il *tipo-restituito* di una funzione è appunto il tipo del valore che viene restituito dalla funzione stessa. Il tipo restituito segue le seguenti regole:

- Le funzioni non possono restituire vettori. Non ci sono restrizioni sul tipo del valore restituito.
- Specificare che il tipo restituito è void indica che la funzione non restituisce alcun valore.

- Se il tipo restituito viene omesso, in C89 si presume che la funzione restituisca un valore di tipo int. Nel C99 non sono ammesse funzioni per le quali è omesso il tipo del valore restituito.

**C99** Alcuni programmatore, per questioni di stile, mettono il valore restituito *sopra* il nome della funzione:

```
double
average(double a, double b)
{
 return (a + b) / 2;
}
```

Mettere il tipo restituito su una riga separata è particolarmente utile quando questo è lungo, come unsigned long int.

**D&R** Dopo il nome della funzione viene messo un elenco di parametri. Ogni parametro viene preceduto da uno specificatore che indica il suo tipo, mentre i diversi parametri sono separati da virgole. Se una funzione non ha nessun parametro allora tra le parentesi deve apparire la parola void. *Nota:* per ogni parametro deve essere specificato il tipo separatamente, anche quando diversi parametri sono dello stesso tipo:

```
double average(double a, b) /** SBAGLIATO **/
{
 return (a + b) / 2;
}
```

Il corpo di una funzione può includere sia dichiarazioni sia istruzioni. Per esempio, la funzione average potrebbe essere scritta come

```
double average(double a, double b)
{
 double sum; /* dichiarazione */
 sum = a + b; /* istruzione */
 return sum / 2; /* istruzione */
}
```

**C99** Le variabili dichiarate nel corpo di una funzione appartengono esclusivamente a quella funzione, non possono essere esaminate o modificate da altre funzioni. Nel C89 la dichiarazione delle variabili deve avvenire prima di tutte le istruzioni presenti nel corpo. Nel C99 invece, dichiarazioni e istruzioni possono essere mischiati fintanto che ogni variabile viene dichiarata precedentemente alla prima istruzione che la utilizza (anche alcuni compilatori pre-C99 permettono di mischiare dichiarazioni e istruzioni).

Il corpo di una funzione il cui tipo restituito è void (che chiameremo una "funzione void") può anche essere vuoto:

```
void print_pun(void)
{
}
```

Lasciare il corpo vuoto può avere senso in fase di sviluppo del programma. Possiamo lasciare uno spazio per la funzione senza perdere tempo a completarla e poi ritornare successivamente a scrivere il corpo.

## Chiamate a funzione

Una chiamata a funzione è costituita dal nome della funzione seguito da un elenco di argomenti racchiusi tra parentesi:

```
average(x, y)
print_count(i)
print_pun()
```

Se mancano le parentesi la funzione non verrà invocata:

 print\_pun;                    /\*\*\* SBAGLIATO \*\*\*/

 Il risultato è un'espressione che, sebbene priva di significato, è ammissibile. L'espressione infatti è corretta ma non ha alcun effetto. Alcuni compilatori emettono un messaggio di warning come *statement with no effect*.

Una chiamata a una funzione void è sempre seguita da un punto e virgola che la trasforma in un'istruzione:

```
print_count(i);
print_pun();
```

Una chiamata a una funzione non-void, invece, produce un valore che può essere memorizzato in una variabile, analizzato, stampato e utilizzato in altri modi:

```
avg = average(x, y);
if (average(x,y) > 0)
 printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```

Nel caso non fosse necessario, il valore restituito da una funzione non-void può sempre essere scartato:

```
average(x, y); /* scarta il valore restituito */
```

Questa chiamata alla funzione average è un esempio di *expression statement*, ovvero di un'istruzione che calcola un'espressione ma che ne scarta il risultato [[expression statement > 4.5](#)].

Ignorare il valore restituito da average può sembrare strano, tuttavia per alcune funzioni ha senso farlo. La funzione printf, per esempio, restituisce il numero di caratteri stampati. Dopo la seguente invocazione la variabile num\_char avrà un valore pari a 9:

```
num_char = printf("Hi, Mom!\n");
```

Dato che di solito non siamo interessati al numero di caratteri stampati dalla funzione, possiamo scartare il valore restituito dalla printf:

```
printf("Hi, Mom!\n"); /* scarta il valore restituito */
```

Per rendere chiaro che stiamo scartando deliberatamente il valore restituito da un'una funzione, il C ci permette di scrivere (void) prima della sua chiamata:

```
(void) printf("Hi, Mom!\n");
```

Quello che stiamo facendo è effettuare un casting (una conversione) del valore restituito dalla printf al tipo void (in C il "casting a void" è un modo educato per dire "getta via") [casting > 7.4]. Usare (void) rende esplicito a tutti che state deliberatamente scartando il valore restituito e non avete semplicemente dimenticato che ce n'era uno. Sfortunatamente c'è una moltitudine di funzioni della libreria del C i cui valori vengono sistematicamente ignorati. Usare (void) tutte le volte che queste funzioni vengono invocate può essere estenuante, per questo motivo all'interno del libro ci asterremo dal farlo.

INTRODUZIONE

## Controllare se un numero è primo

Per vedere come le funzioni possano rendere i programmi più comprensibili, scriviamo un programma che controlli se un numero è primo. Il programma chiede all'utente di immettere un numero e poi risponde con un messaggio indicante se il numero è primo o meno:

Enter a number: 34

Not prime

Invece di mettere i dettagli del controllo all'interno del main, definiamo una funzione separata che restituisce true se il suo parametro è primo e restituisce false se non lo è. Quando le viene dato un numero n, la funzione is\_prime si occupa di dividere n per ogni numero compreso tra 2 e la radice quadrata di n. Se il resto di una delle divisioni è zero, allora sappiamo che il numero non è primo.

```
prime.c /* Controlla se un numero è primo */

#include <stdbool.h> /* solo C99 */
#include <stdio.h>

bool is_prime(int n)
{
 int divisor;

 if (n <= 1)
 return false;
 for (divisor = 2; divisor * divisor <= n; divisor++)
 if (n % divisor == 0)
 return false;
 return true;
}

int main(void)
{
 int n;
```

```

printf("Enter a number: ");
scanf("%d", &n);
if (is_prime(n))
 printf("Prime\n");
else
 printf("Not prime\n");
return 0;
}

```

Osservate come il `main` contenga una variabile chiamata `n` nonostante la funzione `is_prime` abbia un parametro chiamato `n`. In generale, una funzione può dichiarare una variabile con lo stesso nome di una variabile appartenente a un'altra funzione. Le due variabili rappresentano delle locazioni diverse all'interno della memoria e quindi assegnare un nuovo valore a una variabile non modificherà il valore dell'altra (questa proprietà si estende anche ai parametri). La Sezione 10.1 tratta questo argomento in maggiore dettaglio.

Così come dimostra `is_prime`, una funzione può avere più di un'istruzione `return`. Tuttavia durante una chiamata alla funzione solo una di queste istruzioni verrà eseguita. Questo comportamento è la diretta conseguenza del fatto che il raggiungimento di un'istruzione `return` imponga alla funzione il ritorno al punto nella quale era stata chiamata. Impareremo di più sull'istruzione `return` nella Sezione 9.4.

## 9.2 Dichiarazioni di funzioni

Nel programma della Sezione 9.1 la definizione di ogni funzione è sempre stata posta *sopra* il punto nella quale veniva invocata per la prima volta. Agli effetti pratici il C non richiede che la definizione di una funzione preceda le sue chiamate. Supponete di modificare il programma `average.c` mettendo la definizione *dopo* il `main`:

```

#include <stdio.h>

int main(void)
{
 double x, y, z;

 printf("Enter three numbers: ");
 scanf("%lf%lf%lf", &x, &y, &z);
 printf("Average of %g and %g: %g\n", x, y, average(x, y));
 printf("Average of %g and %g: %g\n", y, z, average(y, z));
 printf("Average of %g and %g: %g\n", x, z, average(x, z));

 return 0;
}

double average(double a, double b)
{
 return (a + b) / 2;
}

```

Quando all'interno del `main` il compilatore incontra la prima chiamata alla funzione `average`, non possiede alcuna informazione su quest'ultima: non sa quanti parametri abbia

questa funzione, di che tipo questi siano e nemmeno il tipo del valore restituito. Nonostante questo, invece di produrre un messaggio di errore, il compilatore assume che average ritorni un valore di tipo int (nella Sezione 9.1 abbiamo visto che per default il tipo restituito da una funzione è int). In tal caso diciamo che il compilatore ha creato una **dichiarazione implicita** della funzione. Il compilatore non è in grado di controllare se stiamo passando ad average il giusto numero di argomenti e se questi siano di tipo appropriato. Effettua invece la promozione di default degli argomenti e “spera per il meglio” [**promozione di default degli argomenti > 9.3**]. Quando, più avanti nel programma, incontra la definizione di average, il compilatore prende atto del fatto che il tipo restituito dalla funzione è un double e non un int e quindi otteniamo un messaggio di errore.

Un modo per evitare il problema della chiamata prima della definizione è quello di adattare il programma in modo che la definizione di una funzione preceda tutte le sue invocazioni. Questo adattamento non è sempre possibile purtroppo, e anche quando lo fosse renderebbe il programma difficile da capire perché pone la definizione delle funzioni secondo un ordine innaturale.

Fortunatamente il C offre una soluzione migliore: dichiarare ogni funzione prima di chiamarla. Una **dichiarazione di funzione** fornisce al compilatore un piccolo scorcio della funzione la cui definizione completa verrà fornita successivamente. La dichiarazione di una funzione rispecchia la prima linea della definizione con un punto e virgola aggiunto alla fine:

tipo-restituito nome-funzione (parametri);



Non c’è bisogno di dire che la dichiarazione di una funzione deve essere consistente con la sua definizione.

Ecco come dovrebbe presentarsi il nostro programma con l’aggiunta della dichiarazione di average:

```
#include <stdio.h>

double average(double a, double b); /* DICHIARAZIONE */

int main(void)
{
 double x, y, z;

 printf("Enter three numbers: ");
 scanf("%lf%lf%lf", &x, &y, &z);
 printf("Average of %g and %g: %g\n", x, y, average(x, y));
 printf("Average of %g and %g: %g\n", y, z, average(y, z));
 printf("Average of %g and %g: %g\n", x, z, average(x, z));

 return 0;
}

double average(double a, double b) /* DEFINIZIONE */
{
 return (a + b) / 2;
}
```

D&amp;R

Le dichiarazioni di funzioni del tipo che stiamo discutendo sono conosciute come **prototipi di funzione** per distinguerle dallo stile più vecchio di dichiarazioni dove la parentesi venivano lasciate vuote. Un prototipo fornisce una descrizione completa su come chiamare una funzione: quanti argomenti fornire, di quale tipo debbano essere e quale sarà il tipo restituito.

Per inciso, il prototipo di una funzione non è obbligato a specificare il nome dei parametri, è sufficiente che sia presente il loro tipo:

```
double average(double, double);
```

D&amp;R

In ogni caso di solito non è bene omettere i nomi dei parametri, sia perché questi aiutano a documentare lo scopo di ogni parametro, sia perché ricordano al programmatore l'ordine nel quale questi devono comparire quando la funzione viene chiamata. Tuttavia ci sono delle ragioni legittime per omettere il nome dei parametri e alcuni programmatore preferiscono comportarsi in questo modo.

C99

Il C99 ha adottato la regola secondo la quale prima di tutte le chiamate a una funzione, deve essere presente o la dichiarazione o la definizione della funzione stessa. Chiamare una funzione per la quale il compilatore non ha ancora visto una dichiarazione o una definizione è considerato un errore.

## 9.3 Argomenti

Concentriamoci sulla differenza tra parametri e argomenti. I *parametri* compaiono nelle *definizioni* delle funzioni e sono dei nomi che rappresentano i valori che dovranno essere forniti alla funzione quando questa verrà chiamata. Gli *argomenti* sono delle espressioni che compaiono nelle *chiamate* alle funzioni. A volte, quando la distinzione tra *argomento* e *parametro* non è eccessivamente importante, useremo la parola *argomento* per indicare entrambi.

Nel C gli argomenti vengono **passati per valore**: quando una funzione viene chiamata ogni argomento viene calcolato e il suo valore viene assegnato al parametro corrispondente. Dato che i parametri contengono una copia del valore degli argomenti, ogni modifica apportata ai primi durante l'esecuzione della funzione non avrà alcun effetto sui secondi. Agli effetti pratici ogni parametro si comporta come una variabile che è stata inizializzata con il valore dell'argomento corrispondente.

Il fatto che gli argomenti vengano passati per valore comporta sia vantaggi che svantaggi. Dato che i parametri possono essere modificati senza compromettere i corrispondenti argomenti possiamo usarli come delle variabili interne alla funzione, riducendo così il numero di variabili necessarie. Considerate la seguente funzione che eleva il numero *x* alla potenza *n*:

```
int power(int x, int n)
{
 int i, result = 1;
 for (i = 1; i <= n; i++)
 result = result * x;
 return result;
}
```

Dato che *n* è una *copia* dell'esponente originale, possiamo modificarlo all'interno della funzione eliminando il bisogno della variabile *i*:

```
int power(int x, int n)
{
 int result = 1;
 while (n-- > 0)
 result = result * x;
 return result;
}
```

Purtroppo l'obbligo del C di passare gli argomenti per valore rende difficile scrivere alcuni tipi di funzioni. Per esempio supponete di aver bisogno di una funzione che scomponga un valore double nella sua parte intera e nella sua parte frazionaria. Dato che la funzione non può restituire due numeri, possiamo cercare di passarle una coppia di variabili lasciando a questa il compito di modificarle:

```
void decompose(double x, long int_part, double frac_part)
{
 int_part = (long) x; /* elimina la parte frazionaria di x */
 frac_part = x - int_part;
}
```

Supponete di chiamare la funzione in questo modo:

```
decompose(3.14159, i, d);
```

All'inizio della chiamata 3.14159 viene copiato dentro *x*, il valore di *i* viene copiato dentro *int\_part* e il valore di *d* viene copiato dentro *frac\_part*. Successivamente le istruzioni all'interno della funzione decompose assegnano a *int\_part* il valore 3 e a *frac\_part* il valore 0.14159. Sfortunatamente *i* e *d* non risentono delle assegnazioni a *int\_part* e a *frac\_part*, di conseguenza mantengono il valore che possedevano prima della chiamata anche dopo l'esecuzione di quest'ultima. Come vedremo nella Sezione 11.4, con un po' di lavoro extra è possibile ottenere quanto volevamo dalla funzione *decompose*. Tuttavia, per riuscire a farlo, dobbiamo trattare ancora diverse caratteristiche del C.

## Conversione degli argomenti

Il C permette di effettuare delle chiamate a funzioni dove il tipo degli argomenti non combacia con quello dei parametri. Le regole che governano la conversione degli argomenti dipendono dal fatto che il compilatore abbia visto o meno il prototipo della funzione (o la sua intera definizione) prima della chiamata.

- **Il compilatore ha incontrato il prototipo prima della chiamata.** Il valore di ogni argomento viene implicitamente convertito al tipo del parametro corrispondente così come avverrebbe in un'assegnazione. Per esempio: se un argomento *int* viene passato a una funzione che si aspettava un *double*, l'argomento viene convertito automaticamente al tipo *double*.

- Il compilatore non ha incontrato il prototipo prima della chiamata. Il compilatore esegue le **promozioni di default degli argomenti** (*default argument promotions*): (1) gli argomenti float vengono convertiti in double. (2) Vengono eseguite le promozioni integrali (*integral promotions*) risultando nella conversione al tipo int di tutti gli argomenti char e short.



Affidarsi alle conversioni di default è pericoloso. Considerate il programma seguente:

```
#include <stdio.h>

int main(void)
{
 double x = 3.0;
 printf("Square: %d\n", square(x));

 return 0;
}

int square(int n)
{
 return n * n;
}
```

Quando la funzione square viene chiamata, il compilatore non ne ha ancora visto un prototipo e di conseguenza non sa che square si aspetta un argomento di tipo int. Il compilatore invece esegue su x le promozioni di default degli argomenti, senza che queste portino ad alcun effetto. Considerato che si aspettava un argomento di tipo int mentre ha ricevuto al suo posto un valore double, l'effetto di square non è definito. Il problema può essere risolto effettuando un casting al tipo appropriato sull'argomento di square:

```
printf("Square: %d\n", square((int) x));
```

Naturalmente una soluzione di gran lunga migliore è quella di fornire un prototipo per la funzione square prima che questa venga chiamata. Nel C99 chiamare la funzione square senza fornirne prima una dichiarazione o una definizione è considerato un errore.



## Vettori usati come argomenti

I vettori vengono spesso utilizzati come argomento. Quando il parametro di una funzione è costituito da un vettore unidimensionale, la lunghezza del vettore può non essere specificata (e di solito non lo è):

```
int f(int a[]) /* nessuna lunghezza specificata */
{
 -
}
```

L'argomento può essere un vettore unidimensionale i cui elementi siano del tipo appropriato. C'è solamente un problema: come farà la funzione f a determinare la lunghezza del vettore. Sfortunatamente il C non prevede per le funzioni un modo semplice per determinare la lunghezza di un vettore che viene passato come argo-

mento. Invece, se la funzione ne ha bisogno, dovremo fornire la lunghezza noi stessi come un ulteriore argomento.



Sebbene per cercare di determinare la lunghezza di una *variabile* vettore sia possibile usare l'operatore `sizeof`, questo non fornisce il risultato corretto per un *parametro* vettore:

```
int f(int a[])
{
 int len = sizeof(a) / sizeof(a[0]);
 /* SBAGLIATO: non è il numero di elementi di a */
}
```

La Sezione 12.3 spiega il perché.

La funzione seguente illustra l'uso di vettori unidimensionali come argomenti. Quando le viene passato un vettore `a` di valori `int`, la funzione `sum_array` restituisce la somma degli elementi presenti in `a`. Considerato che `sum_array` ha bisogno di conoscere quale sia la lunghezza di `a`, dobbiamo fornire un secondo argomento.

```
int sum_array(int a[], int n)
{
 int i, sum = 0;
 for (i = 0; i < n; i++)
 sum += a[i];
 return sum;
}
```

Il prototipo di `sum_array` ha la seguente forma:

```
int sum_array(int a[], int n);
```

Come al solito, se lo vogliamo, possiamo omettere il nome dei parametri:

```
int sum_array(int [], int);
```

Quando `sum_array` viene chiamata, il primo argomento è il nome del vettore mentre il secondo la sua lunghezza. Per esempio:

```
#define LEN 100
int main(void)
{
 int b[LEN], total;
 total = sum_array(b, LEN);
}
```

Notate che quando un vettore viene passato a una funzione, a seguito del suo nome non vengono messe le parentesi quadre.

```
total = sum_array(b[], LEN); /** SBAGLIATO **/
```

Una questione che riguarda i vettori usati come argomenti è che una funzione non ha modo di controllare se le abbiamo passato la loro lunghezza corretta. Possiamo accertaci di questo dicendo alla funzione che il vettore è più piccolo di quello che è in realtà. Supponete di aver salvato solamente 50 numeri nel vettore b anche se questo può contenerne 100. Possiamo sommare solo i primi 50 elementi scrivendo:

```
total = sum_array(b, 50); /* somma i primi 50 elementi */
```

la funzione sum\_array ignorerà gli altri 50 elementi (non saprà nemmeno che esistono!).



Fate attenzione a non dire a una funzione che il vettore è più *grande* di quello che è in realtà:

```
total = sum_array(b, 150); /* SBAGLIATO */
```

In questo esempio la funzione sum\_array oltrepasserà la fine del vettore causando un comportamento indefinito.

Un'altra cosa importante da sapere è che a una funzione è permesso modificare gli elementi di un vettore passato come parametro e che la modifica si riperquo sul-l'argomento corrispondente. La seguente funzione, per esempio, modifica un vettore memorizzando uno zero in ognuno dei suoi elementi:

```
void store_zeros(int a[], int n)
{
 int i;
 for (i = 0; i < n; i++)
 a[i] = 0;
}
```

La chiamata

```
store_zeros(b, 100);
```

memorizzerà uno zero nei primi 100 elementi del vettore b. La possibilità di modificare un vettore passato come argomento sembra contraddirre il fatto che il C passi gli argomenti per valore. In effetti non c'è alcuna contraddizione, ma non potremo capirlo fino alla Sezione 12.3.



Se un parametro è costituito da un vettore multidimensionale, nella dichiarazione del parametro può essere omessa solo la prima dimensione. Per esempio, se modifichiamo la funzione sum\_array in modo che a sia un vettore a due dimensioni, allora dobbiamo specificare il numero delle colonne di a, anche se non abbiamo indicato il numero di righe:

```
#define LEN 10
int sum_two_dimensional_array(int a[][LEN], int n)
{
 int i, j, sum = 0;
```

```

 for (i = 0; i < n; i++)
 for (j = 0; j < n; j++)
 sum += a[i][j];
 return sum;
}

```

Non essere in grado di passare vettori multidimensionali con un numero arbitrario di colonne può essere una seccatura. Fortunatamente, molto spesso possiamo aggirare questo problema usando dei vettori di puntatori [[vettori di puntatori > 13.7](#)]. I vettori a lunghezza variabile del C99 forniscono una soluzione al problema ancora migliore.

## Vettori a lunghezza variabile usati come argomenti

Il C99 aggiunge diverse novità ai vettori usati come argomenti. La prima ha a che fare con i vettori a lunghezza variabile (VLA) [[vettori a lunghezza variabile > 8.3](#)], una funzionalità del C99 che permette di specificare la lunghezza di un vettore per mezzo un'espressione non costante. Naturalmente anche i vettori a lunghezza variabile possono essere usati come parametri.

Considerate la funzione `sum_array` che è stata discussa nella sezione precedente. Ecco la definizione di `sum_array` alla quale è stato omesso il corpo:

```

int sum_array(int a[], int n)
{
 ...
}

```

Per com'è adesso, non c'è alcun legame diretto tra `n` e la lunghezza del vettore `a`. Sebbene il corpo della funzione tratti `n` come la lunghezza di `a`, la vera lunghezza del vettore può essere maggiore di `n` (o minore, nel qual caso la funzione non funzionerebbe a dovere).

Usando un vettore a lunghezza variabile come parametro, possiamo indicare specificamente che `n` è la lunghezza di `a`:

```

int sum_array(int n, int a[n])
{
 ...
}

```

Il valore del primo parametro (`n`) indica la lunghezza del secondo parametro (`a`). Osservate come l'ordine dei parametri sia stato invertito. L'ordine è importante quando vengono usati i vettori a lunghezza variabile.

---

 La seguente versione di `sum_array` non è ammissibile:

```

int sum_array(int a[n], int n) /** SBAGLIATO ***
{
 ...
}

```

Il compilatore emetterà un messaggio di errore quando incontra `int a[n]` a causa del fatto che non ha ancora visto `n`.

---

Ci sono diversi modi per scrivere il prototipo della nostra nuova versione di `sum_array`. Una possibilità è che segua esattamente la definizione delle funzione:

```
int sum_array(int n, int a[n]); /* Versione 1 */
```

Un'altra possibilità è quella di rimpiazzare la lunghezza del vettore con un asterisco (\*):

```
int sum_array(int n, int a[*]); /* Versione 2a */
```

La ragione di usare la notazione con l'asterisco è che i nomi dei parametri sono opzionali nelle dichiarazioni delle funzioni. Se il nome del primo parametro viene omesso, non sarà possibile specificare che la lunghezza del vettore sia `n`. L'asterisco indica che la lunghezza del vettore è collegata ai parametri che lo precedono nella lista:

```
int sum_array(int, int [*]); /* Versione 2b */
```

È permesso anche lasciare le parentesi quadre vuote, esattamente come quando dichiariamo normalmente un vettore come parametro:

```
int sum_array(int n, int a[]); /* Versione 3a */
int sum_array(int, int []); /* Versione 3b */
```

Lasciare la parentesi vuote non è una buona scelta perché non esplicita la relazione tra `n` e `a`.

In generale, una qualsiasi espressione può essere usata come lunghezza di un parametro costituito da un vettore a lunghezza variabile. Supponete per esempio di scrivere una funzione che concateni due vettori `a` e `b` copiando gli elementi di `a` nel vettore `c` e facendoli poi seguire dagli elementi di `b`:

```
int concatenate(int m, int n, int a[m], int b[n], int c[m+n])
{
 ...
}
```

La lunghezza del vettore `c` sarà uguale alla somma delle lunghezze di `a` e `b`. L'espressione utilizzata per specificare la lunghezza di `c` coinvolge altri due parametri, ma in generale può fare riferimento a delle variabili presenti al di fuori della funzione o persino chiamare altre funzioni.

I vettori a lunghezza variabile di una sola dimensione (come quelli degli esempi visti finora) hanno un'utilità limitata. Rendono la dichiarazione di una funzione o la sua definizione più descrittiva indicando la lunghezza desiderata per un argomento costituito da un vettore. Tuttavia non viene eseguito nessun controllo aggiuntivo di eventuali errori, infatti per un vettore usato come argomento è ancora possibile essere troppo lungo o troppo corto.

Ne risulta che i parametri costituiti da vettori a lunghezza variabile sono più utili per i vettori multidimensionali. Precedentemente in questa sezione abbiamo cercato di scrivere una funzione che somma gli elementi di un vettore a due dimensioni. La nostra funzione originale era limitata ai vettori con un numero di colonne prefissato. Se come parametro usiamo un vettore a lunghezza variabile allora possiamo generalizzare la funzione a un qualsiasi numero di colonne.

```

int sum_two_dimensional_array(int n, int m, int a[n][m])
{
 int i, j, sum = 0;
 for (i = 0; i < n; i++)
 for (j = 0; j < m; j++)
 sum += a[i][j];
 return sum;
}

```

I seguenti possono tutti essere dei prototipi per la funzione appena vista:

```

int sum_two_dimensional_array(int n, int m, int a[n][m]);
int sum_two_dimensional_array(int n, int m, int a[*][*]);
int sum_two_dimensional_array(int n, int m, int a[][m]);
int sum_two_dimensional_array(int n, int m, int a[][*]);

```

## C99 Usare static nella dichiarazione di un parametro vettore

Il C99 ammette l'uso della parola chiave `static` nella dichiarazione di parametri vettore (la stessa keyword esisteva prima del C99. La Sezione 18.2 discute dei suoi usi tradizionali).

Nell'esempio seguente, l'aver posto `static` davanti al numero 3 indica che si garantisce che la lunghezza di `a` sia almeno pari a 3:

```

int sum_array(int a[static 3], int n)
{
 ...
}

```

Usare `static` in questo modo non ha alcun effetto sul comportamento del programma. La presenza di `static` è un semplice suggerimento che permette al compilatore di generare delle istruzioni più veloci per l'accesso al vettore (se il compilatore sa che il vettore avrà sempre un certa lunghezza minima può "pre-caricare" quegli elementi dalla memoria nel momento in cui la funzione viene invocata e quindi prima che gli elementi siano effettivamente necessari alle istruzioni).

Ancora una nota a riguardo a questa keyword: se un parametro vettore ha più di una dimensione, allora `static` può essere usata solamente per la prima di queste dimensioni (per esempio, quando si specifica il numero di righe in un vettore bidimensionale).

## C99 Letterali composti

Ritorniamo un'ultima volta sulla versione originale della funzione `sum_array`. Quando `sum_array` viene chiamata, di solito il primo argomento è il nome del vettore (quello i cui elementi verranno sommati). Per esempio, possiamo chiamare `sum_array` nel modo seguente:

```
int b[] = {3, 0, 3, 4, 1};
```

```
total = sum_array(b, 5);
```

Questo metodo presenta un unico problema, ovvero `b` deve essere dichiarato come una variabile ed essere inizializzata prima di effettuare la chiamata. Se `b` non fosse necessaria a nessun altro scopo, sarebbe piuttosto sgradevole doverla creare solo per effettuare una chiamata a `sum_array`.

Nel C99 possiamo evitare questa seccatura usando un **letterale composto** (*compound literal*): un vettore senza nome che viene creato al volo specificando semplicemente gli elementi che contiene. La chiamata seguente alla funzione `sum_array` contiene un letterale composto (indicato in grassetto) come primo argomento:

```
total = sum_array((int []){3, 0, 3, 4, 1}, 5);
```

In questo esempio il letterale composto crea un vettore con cinque interi: 3, 0, 3, 4 e 1. Non abbiamo specificato la lunghezza del vettore e quindi questa viene determinata dal numero di elementi presenti. Opzionalmente possiamo anche specificare in modo esplicito la lunghezza del vettore: `(int [4]){1, 9, 2, 1}` che è equivalente a `(int []){1, 9, 2, 1}`.

In generale un letterale composto consiste del nome di un tipo racchiuso tra parentesi tonde, seguito da un insieme di valori racchiusi tra parentesi graffe. Un letterale composto rispecchia un cast applicato a un inizializzatore, infatti i letterali composti e gli inizializzatori obbediscono alle stesse regole. Come un inizializzatore designato [**inizializzatori designati > 8.1**], anche un letterale composto può contenere un designatore e allo stesso modo può evitare di fornire l'inizializzazione completa (in tal caso gli elementi non inizializzati vengono tutti posti a zero). Per esempio, il letterale `(int [10]){8, 6}` ha 10 elementi, i primi due hanno i valori 8 e 6 mentre gli altri hanno valore 0.

I letterali composti creati all'interno di una funzione possono contenere una qualsiasi espressione. Per esempio possiamo scrivere

```
total = sum_array((int []){2 * i, i + j, j * k}, 3);
```

dove `i`, `j` e `k` sono delle variabili. Questo aspetto dei letterali composti accresce di molto la loro utilità.

Un letterale composto è un lvalue e quindi i valori dei suoi elementi possono essere modificati [**lvalues > 4.2**]. Se lo si desidera un letterale composto può essere impostato in "sola lettura" aggiungendo la parola `const` al suo tipo come in `(const int []){5, 4}`.

## 9.4 L'istruzione return

Una funzione non `void` deve usare l'istruzione `return` per specificare il valore che sarà restituito. L'istruzione `return` ha il seguente formato:

```
return espressione;
```

Spesso l'espressione è costituita solamente da una costante o da una variabile:

```
return 0;
```

```
return status;
```

Sono possibili espressioni più complesse. Per esempio non è raro vedere l'operatore condizionale [operatore condizionale > 5.2] usato in abbinamento all'istruzione return:

```
return n >= 0 ? n : 0;
```

Quando questa istruzione viene eseguita, per prima cosa viene calcolata l'espressione  $n \geq 0 ? n : 0$ . L'istruzione restituisce il valore di  $n$  se questo non è negativo, altrimenti restituisce uno 0.

Se il tipo dell'espressione di un'istruzione return non combacia con il tipo restituito dalla funzione, questo viene implicitamente convertito al tipo adeguato. Per esempio, se viene dichiarato che una funzione restituisce un int ma l'istruzione return contiene un'espressione double, allora il valore dell'espressione viene convertito in int.

L'espressione return può anche comparire in funzioni il cui tipo restituito è void, ammesso che non venga fornita nessuna espressione:

```
return; /* return in una funzione void */
```



Mettere un'espressione in questa istruzione return comporterebbe un errore all'atto della compilazione. Nell'esempio seguente, l'istruzione return fa sì che la funzione termini immediatamente quando le viene fornito un argomento negativo:

```
void print_int(int i)
{
 if (i < 0)
 return;
 printf("%d", i);
}
```

Se i è minore di 0 allora la funzione print\_int terminerà senza chiamare la printf.

Un'istruzione return può comparire anche alla fine di una funzione void:

```
void print_pun(void)
{
 printf("To C, or not to C: that is the question.\n");
 return; /* va bene, ma non è necessario */
}
```

Usare return non è necessario dato che la funzione ritornerebbe automaticamente dopo l'esecuzione della sua ultima istruzione.

Se una funzione non-void raggiunge la fine del suo corpo (cioè senza eseguire l'istruzione return), il comportamento del programma non risulterebbe definito qualora quest'ultimo cercasse di utilizzare il valore restituito dalla funzione. Alcuni compilatori possono generare un messaggio di warning come *control reaches end of non-void function* se rilevano la possibilità che una funzione non-void fuoriesca dal suo corpo.

## 9.5 Interrompere l'esecuzione di un programma

Dato che è una funzione, anche il `main` deve avere un tipo restituito. Normalmente il tipo restituito dal `main` è `int` e questo è il motivo per il quale finora abbiamo definito il `main` come segue:

```
int main(void)
{
}
```

I programmi C più vecchi omettono il tipo restituito dal `main` avvantaggiandosi del fatto che tradizionalmente è considerato `int` per default:

```
main()
{
}
```

 È meglio evitare questa pratica dato che nel C99 l'omissione del tipo restituito non viene ammessa. Omettere la parola `void` nella lista di parametri del `main` è ammesso, ma (per ragioni di stile) è meglio essere esplicativi nel definire che il `main` non possiede parametri (vedremo più avanti che a volte il `main` ha dei parametri che di solito vengono chiamati `argc` e `argv` [\[argc e argv > 13.7\]](#)).

 Il valore restituito dal `main` è un codice di stato che (in alcuni sistemi operativi) può essere testato al termine del programma. Il `main` dovrebbe restituire uno 0 se il programma termina normalmente, mentre per indicare una fine anomala il `main` dovrebbe restituire un valore diverso da zero (in effetti non c'è nessuna regola che ci vietи di utilizzare il valore restituito per altri scopi). È buona pratica assicurarsi che ogni programma C restituisca un codice di stato, anche quando il suo utilizzo non è previsto, perché chi eseguirà il programma potrebbe decidere di analizzarlo.

### La funzione exit

Eseguire un'istruzione `return` è solo uno dei modi per terminare un programma. Un altro è quello di chiamare la funzione `exit` che appartiene all'header `<stdlib.h>` [\[header <stdlib.h> > 26.2\]](#). L'argomento che viene passato a `exit` ha lo stesso significato del valore restituito dal `main`: entrambi indicano lo stato del programma al suo termine. Per indicare che il programma è terminato normalmente passiamo il valore 0:

```
exit(0); /* programma terminato normalmente */
```

Dato che lo 0 è un po' criptico, il C permette di passare al suo posto la macro `EXIT_SUCCESS` (l'effetto è il medesimo):

```
exit(EXIT_SUCCESS); /* programma terminato normalmente */
```

Passare `EXIT_FAILURE` indica invece che il programma è terminato in modo anomale:

```
exit(EXIT_FAILURE); /* programma terminato in modo anomale */
```

`EXIT_SUCCESS` ed `EXIT_FAILURE` sono due macro definite in `<stdlib.h>`.

Il valore di `EXIT_SUCCESS` e di `EXIT_FAILURE` è definito dall'implementazione, i valori tipici sono rispettivamente 0 e 1.

Come metodi per terminare un programma, `return` ed `exit` sono in stretta relazione. Infatti nel `main` l'istruzione

`return espressione;`

è equivalente a

`exit(espressione);`

La differenza tra `return` ed `exit` è che `exit` causa la fine del programma indipendentemente da quale funzione sia a effettuare l'invocazione. L'istruzione `return` causa la fine del programma solo quando appare nella funzione `main`. Alcuni programmatore usano `exit` esclusivamente per rendere facile l'individuazione dei punti di uscita del programma.

## 9.6 Ricorsione

Una funzione è **ricorsiva** se chiama se stessa. La funzione seguente, per esempio, calcola  $n!$  in modo ricorsivo usando la formula  $n! = n \times (n - 1)!$ :

```
int fact(int n)
{
 if (n <= 1)
 return 1;
 else
 return n * fact(n - 1);
}
```

Alcuni linguaggi di programmazione fanno un uso pesante della ricorsione, mentre altri non la permettono nemmeno. Il C ricade da qualche parte nel mezzo di queste due categorie: ammette la ricorsione, ma la maggior parte dei programmi non la usa molto spesso.

Per vedere come agisce la ricorsione, tracciamo l'esecuzione dell'istruzione

`i = fact(3);`

Ecco cosa succede:

`fact(3)` trova che 3 non è minore o uguale a 1 e quindi esegue la chiamata

`fact(2)`, la quale trova che 2 non è minore o uguale a 1 e quindi esegue la chiamata

`fact(1)`, la quale trova che 1 è minore o uguale a 1 e quindi restituisce un 1, così facendo

`fact(2)` restituisce  $2 \times 1 = 2$ , questo comporta che

`fact(3)` restituisca  $3 \times 2 = 6$ .

Osservate come le chiamate non terminate di `fact` si "impilino" fino a quando alla funzione `fact` non viene passato un 1. A quel punto le vecchie chiamate a `fact` ini-

ziano a "srotolarsi" una a una fino a quando la chiamata originale (`fact(3)`) restituisce il risultato.

Ecco un altro esempio di ricorsione: una funzione che calcola  $x^n$  usando la formula  $x^n = x \times x^{n-1}$ .

```
int power(int x, int n)
{
 if (n == 0)
 return 1;
 else
 return x * power(x, n - 1);
}
```

La chiamata `power(5, 3)` verrebbe eseguita come segue:

`power(5, 3)` trova che 3 è diverso da 0 e quindi esegue la chiamata  
`power(5, 2)` la quale trova che 2 non è uguale a 0 e quindi esegue la chiamata  
`power(5, 1)` la quale trova che 1 non è uguale a 0 e quindi esegue la chiamata  
`power(5, 0)` trova che 0 è uguale a 0 e quindi restituisce un 1, facendo sì che  
`power(5, 1)` ritorni  $5 \times 1 = 5$ , questo a sua volta fa sì che  
`power(5, 2)` ritorni  $5 \times 5 = 25$ , questo a sua volta fa sì che  
`power(5, 3)` ritorni  $5 \times 25 = 125$ .

Tra l'altro possiamo condensare la funzione `power` scrivendo un'espressione condizionale nell'istruzione `return`:

```
int power(int x, int n)
{
 return n == 0 ? 1 : x * power(x, n - 1);
}
```

Sia `fact` che `power` sono attente a testare una "condizione di terminazione" appena vengono invocate. Quando viene chiamata, `fact` controlla immediatamente se il suo parametro è minore o uguale a 1. Quando viene invocata `power`, questa controlla se il suo secondo parametro è uguale a 0. Tutte le funzioni ricorsive hanno bisogno di una qualche condizione di termine per evitare una ricorsione infinita.

## Algoritmo Quicksort

A questo punto potreste chiedervi perché ci stiamo preoccupando della ricorsione: dopo tutto né la funzione `fact` né la funzione `power` ne hanno realmente bisogno. Bene, siete arrivati al nocciolo della questione. Nessuna delle due funzioni fa molto caso alla ricorsione perché entrambe chiamano se stesse una volta sola. La ricorsione è molto più utile per algoritmi più sofisticati che richiedono a una funzione di invocare se stessa due o più volte.

Nella pratica la ricorsione nasce spesso come risultato di una tecnica algoritmica conosciuta come **divide-et-impera**, nella quale un problema più grande viene diviso in parti più piccole che vengono affrontate dallo stesso algoritmo. Un esempio classico di questa strategia può essere trovato nel popolare algoritmo di ordinamento chiamato **Quicksort**. L'algoritmo Quicksort funziona in questo modo (per sempli-

ciò assumeremo che il vettore che deve essere ordinato abbia indici che vanno da 1 a  $n$ :

1. Si sceglie un elemento  $e$  del vettore (l'“elemento di partizionamento”) e si sistema il vettore in modo che gli elementi  $1, \dots, i-1$  siano minori o uguali a  $e$ , l’elemento  $i$  contenga  $e$  e che gli elementi  $i+1, \dots, n$  siano maggiori o uguali a  $e$ .
2. Si ordinano gli elementi  $1, \dots, i-1$  usando ricorsivamente l’algoritmo Quicksort.
3. Si ordinano gli elementi  $i+1, \dots, n$  usando ricorsivamente l’algoritmo Quicksort.

Dopo lo step 1, l’elemento  $e$  si trova nella locazione giusta. Dato che gli elementi alla sinistra di  $e$  sono tutti minori o uguali a esso, si troveranno nel posto giusto dopo essere stati ordinati nello step 2. Un ragionamento analogo si applica agli elementi alla destra di  $e$ .

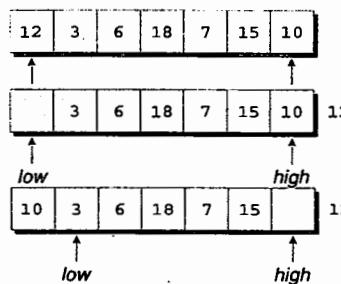
Ovviamente lo step 1 dell’algoritmo Quicksort è critico. Ci sono diversi modi per partizionare un vettore e alcuni sono migliori degli altri. Useremo una tecnica che è facile da capire anche se non particolarmente efficiente. Prima descriveremo l’algoritmo di partizionamento in modo informale e successivamente lo tradurremo in codice C.

L’algoritmo si basa su due “indicatori” chiamati *low* e *high*, che tengono traccia di alcune posizioni all’interno del vettore. Inizialmente il puntatore *low* punta al primo elemento del vettore mentre *high* all’ultimo. Iniziamo copiando il primo elemento (l’elemento di partizionamento) in una locazione temporanea, lasciando un “buco” nel vettore. Poi spostiamo *high* attraversando il vettore da destra a sinistra fino a quando non punta a un elemento che è minore dell’elemento di partizionamento. Successivamente copiamo questo elemento nel buco puntato da *low* creando così un nuovo buco (puntato da *high*). Adesso spostiamo *low* da sinistra a destra cercando un elemento che è maggiore di quello di partizionamento. Quando ne abbiamo trovato uno lo copiamo nel buco al quale punta *high*. Il processo si ripete con *low* e *high* che si danno il cambio fino a quando questi non si incontrano in qualche punto nel mezzo del vettore. In quel momento entrambi puntano allo stesso buco e tutto quello che dobbiamo fare è copiarvi l’elemento di partizionamento. Lo schema seguente illustra come un vettore di interi verrebbe ordinato da Quicksort:

Iniziamo con un vettore contenente sette elementi. *low* punta al primo elemento, *high* punta all’ultimo.

Il primo elemento, 12, è l’elemento di partizionamento. Copiarlo in qualche altro posto lascia un buco all’inizio del vettore.

Adesso confrontiamo l’elemento puntato da *high* con 12. Dato che 10 è minore di 12 questo significa che si trova nel lato sbagliato del vettore e quindi lo spostiamo nel buco e trasliamo *low* verso destra.



*low* punta al numero 3 che è minore di 12 e quindi non ha bisogno di essere spostato. Trasliamo invece *low* verso destra.

Dato che anche 6 è minore di 12 trasliamo *low* un'altra volta.

Adesso *low* punta a 18 che è maggiore di 12 e quindi si trova nella posizione sbagliata. Dopo aver spostato 18 nel buco, trasliamo *high* verso sinistra.

*high* punta a 15 che è maggiore di 12 e quindi non ha bisogno di essere spostato. Trasliamo *high* verso sinistra e continuiamo.

*high* punta a 7 e quindi si trova fuori posto. Dopo aver spostato 7 nel buco, trasliamo *low* a destra.

Adesso *low* e *high* sono uguali e quindi spostiamo l'elemento di partizionamento all'interno del buco.

A questo punto abbiamo raggiunto il nostro obiettivo: tutti gli elementi a sinistra dell'elemento di partizionamento sono minori o uguali a 12, e tutti gli elementi a destra sono maggiori o uguali a 12. Adesso che il vettore è stato partizionato possiamo applicare ricorsivamente Quicksort per ordinare i primi quattro elementi del vettore (10, 3, 6 e 7) e gli ultimi due (15 e 18).

#### PROGRAMMA

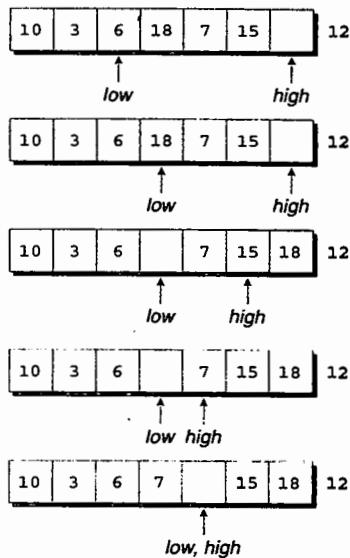
## Quicksort

Sviluppiamo una funzione ricorsiva chiamata quicksort che usi l'algoritmo Quicksort per ordinare un vettore di numeri interi. Per testare la funzione scriviamo un main che legga 10 numeri inserendoli in un vettore, chiama la funzione quicksort per ordinare il vettore e poi stampi gli elementi di quest'ultimo:

```
Enter 10 numbers to be sorted: 9 16 47 82 4 66 12 3 25 51
In sorted order: 3 4 9 12 16 25 47 51 66 82
```

Dato che il codice per il partizionamento del vettore è piuttosto lungo, è stato messo in una funzione separata chiamata split.

```
qsrt.c /* Ordina un vettore di numeri interi usando l'algoritmo Quicksort */
#include <stdio.h>
#define N 10
void quicksort(int a[], int low, int high);
int split(int a[], int low, int high);
```



|    |   |   |   |     |      |    |    |
|----|---|---|---|-----|------|----|----|
| 10 | 3 | 6 | 7 |     | 15   | 18 | 12 |
|    |   |   |   | low | high |    |    |

|    |   |   |   |     |      |    |    |
|----|---|---|---|-----|------|----|----|
| 10 | 3 | 6 | 7 | 12  | 15   | 18 | 12 |
|    |   |   |   | low | high |    |    |

```

int main(void)
{
 int a[N], i;

 printf("Enter %d numbers to be sorted: ", N);
 for (i = 0; i < N; i++)
 scanf("%d", &a[i]);
 quicksort(a, 0, N - 1);

 printf("In sorted order: ");
 for (i = 0; i < N; i++)
 printf("%d ", a[i]);
 printf("\n");

 return 0;
}

void quicksort(int a[], int low, int high)
{
 int middle;

 if (low >= high) return;
 middle = split(a, low, high);
 quicksort(a, low, middle - 1);
 quicksort(a, middle + 1, high);
}

int split(int a[], int low, int high)
{
 int part_element = a[low];

 for (;;) {
 while (low < high && part_element <= a[high])
 high--;
 if (low >= high) break;
 a[low++] = a[high];

 while (low < high && a[low] <= part_element)
 low++;
 if (low >= high) break;
 a[high--] = a[low];
 }

 a[high] = part_element;
 return high;
}

```

Sebbene questa versione di Quicksort funzioni, non è il massimo. Ci sono diversi modi per migliorare le performance del programma, tra cui:

- **Migliorare l'algoritmo di partizionamento.** Il nostro metodo non è il più efficiente possibile. Invece di scegliere il primo elemento del vettore come elemento di partizionamento, è meglio prendere la mediana tra il primo elemento,

quello di mezzo e l'ultimo. Anche lo stesso processo di partizionamento può essere velocizzato. In particolare è possibile evitare il test `low < high` presente nei due cicli `while`.

- **Usare un metodo diverso per ordinare i vettori più piccoli.** Invece di usare ricorsivamente Quicksort fino ai vettori di un elemento, sarebbe meglio usare un metodo più semplice per i vettori più piccoli (diciamo quelli con meno di 25 elementi).
- **Rendere Quicksort non ricorsiva.** Sebbene Quicksort sia per sua natura un algoritmo ricorsivo, e sia più facile da capire nella sua forma ricorsiva, in effetti risulta più efficiente se la ricorsione viene eliminata.

## Domande & Risposte

**D:** Alcuni libri sul C usano termini diversi da *parametro* e *argomento*. Esiste una terminologia standard? [p. 192]

**R:** Così come in altri aspetti del C non c'è un accordo generale sulla terminologia, sebbene gli standard C89 e C99 usano i termini *parametro* e *argomento*. La tabella seguente dovrebbe aiutarvi nelle traduzioni:

| <i>Questo libro:</i> | <i>Altri libri:</i>                  |
|----------------------|--------------------------------------|
| parametro            | argomento formale, parametro formale |
| argomento            | argomento attuale, parametro attuale |

Tenete a mente che, quando non c'è pericolo di creare confusione, sfumeremo intenzionalmente la distinzione tra i due termini usando la parola *argomento* per indicare entrambi.

**D:** Abbiamo visto dei programmi nei quali i tipi sono specificati in dichiarazioni separate poste dopo la lista dei parametri, così come succede nell'esempio seguente:

```
double average(a, b)
double a, b;
{
 return (a + b) / 2;
}
```

**Questa pratica è permessa?** [p. 196]

**R:** Questo modo di definire le funzioni deriva dal K&R C e quindi potete incontrarlo nei vecchi libri di programmazione. Il C89 e il C99 supportano questo stile in modo che i vecchi programmi possano essere ancora compilati. Tuttavia è meglio evitarne l'uso nei nuovi programmi per un paio di ragioni.

Per prima cosa le funzioni che vengono definite nel vecchio stile non sono sogrette allo stesso grado di controllo degli errori. Quando una funzione viene definita nella vecchia maniera (e il prototipo non è presente) il compilatore non controlla se quella funzione viene chiamata con il numero corretto di elementi e non controlla nemmeno se gli argomenti sono del tipo appropriato. Esegirà invece le promozioni di default degli argomenti [**promozioni di default degli argomenti > 9.2**].

Secondariamente lo standard C afferma che il vecchio stile è “obsoleto”, intendendo che il suo utilizzo viene scoraggiato e che in futuro potrebbe anche essere escluso dal C.

**D: Alcuni linguaggi di programmazione permettono a procedure e funzioni di annidarsi le une dentro le altre. Il C permette di annidare delle definizioni di funzioni?**

**R:** No, il C non ammette che la definizione di una funzione venga annidata nel corpo di un'altra funzione. Questa restrizione, tra le altre cose, semplifica il compilatore.

**\*D: Perché il compilatore permette di usare dei nomi di funzione che non sono seguiti dalle parentesi? [p. 197]**

**R:** Vedremo in un capitolo più avanti che il compilatore tratta un nome di funzione non seguito da parentesi come un puntatore alla funzione [**puntatori a funzione > 17.7**]. I puntatori alle funzioni hanno degli usi consentiti e quindi il compilatore non può assumere automaticamente che il nome di una funzione senza le parentesi sia un errore. L'istruzione

```
print_pun;
```

è ammessa perché il compilatore tratta `print_pun` come un puntatore e questo rende l'istruzione un expression statement valido [**expression statement > 4.5**], sebbene privo di senso.

**\*D: Nella chiamata a funzione `f(a, b)`, come fa il compilatore a sapere se la virgola è un separatore o un operatore?**

**R:** In effetti gli argomenti delle chiamate a funzione non possono essere delle espressioni qualsiasi. Infatti devono essere delle “espressioni di assegnamento” che non possono contenere delle virgolette usate come operatori a meno che queste non vengano racchiuse da delle parentesi. In altre parole, la virgola nella chiamata `f(a, b)` è un separatore mentre nella chiamata `f((a, b))` è un operatore.

**D: I nomi dei parametri nel prototipo di una funzione devono coincidere con quelli forniti successivamente dalla definizione? [p. 200]**

**R:** No. Alcuni programmatore sfruttano questo fatto dando lunghi nomi nel prototipo e usando dei nomi più corti nella definizione. Per esempio un programmatore francofono potrebbe utilizzare nomi inglesi nei prototipi per poi passare a dei nomi francesi nella definizione della funzione.

**D: Non capiamo ancora perché ci si deve preoccupare dei prototipi delle funzioni. Se mettiamo tutte le definizioni prima del `main` non è tutto a posto?**

**R:** No. Per prima cosa state assumendo che solo il `main` chiama altre funzioni, il che è irrealistico. Nella pratica infatti alcune funzioni si chiameranno tra loro. Se mettiamo tutte le definizioni sopra il `main` dobbiamo fare attenzione a ordinarle accuratamente. Chiamare una funzione che non è stata ancora definita può comportare dei seri problemi.

Non è tutto però. Supponete che due funzioni si chiamino l'un l'altra (il che non è così strano come possa sembrare). Indipendentemente da quale funzione viene definita per prima, finiamo sempre per invocare una funzione che non è stata definita.

Ma c'è dell'altro! Una volta che i programmi raggiungono una certa dimensione non è praticabile mettere tutte le funzioni all'interno dello stesso file. Quando raggiungiamo quel punto, abbiamo la necessità che i prototipi delle funzioni informino il compilatore delle funzioni presenti negli altri file.

**D: Abbiamo visto delle dichiarazioni che omettono tutte le informazioni sui parametri:**

```
double average();
```

**questa pratica viene ammessa? [p. 201]**

**R:** Sì. Questa dichiarazione informa il compilatore che la funzione average restituisce un double, ma non fornisce alcuna informazione sul numero e sul tipo dei suoi parametri (lasciare le parentesi vuote non significa necessariamente che average non abbia parametri).

Nel C di K&R, questa è l'unica forma ammessa per le dichiarazioni. Il formato che stiamo usando nel libro (quello con il prototipo della funzione dove le informazioni sui parametri vengono incluse) è stato introdotto con il C89. Oggi il vecchio tipo di dichiarazione, anche se ammesso, è considerato obsoleto.

**D: Perché un programmatore dovrebbe omettere deliberatamente i nomi dei parametri nel prototipo di una funzione? Non è più semplice mantenerli? [p. 201]**

**R:** L'omissione dei nomi dei parametri di solito viene fatta per scopi di difesa. Se succede che una macro abbia lo stesso nome di un parametro, questo nome verrà rimpiazzato durante il preprocessamento, danneggiando di conseguenza il prototipo. Di solito questo non è un problema nei piccoli programmi scritti da una sola persona, ma può accadere in grandi applicazioni scritte da più persone.

**D: È possibile mettere la dichiarazione di una funzione all'interno del corpo di un'altra funzione?**

**R:** Sì, ecco un esempio:

```
int main(void)
{
 double average(double a, double b);
}
```

Questa dichiarazione di average è valida solo per il corpo del main. Se altre funzioni devono invocare average, allora ognuna di esse deve dichiararla.

Il vantaggio di questa tecnica è che diventa chiaro per il lettore capire quali funzioni chiamano le altre (in questo esempio vediamo che il main chiamerà average). D'altro canto può essere una seccatura nel caso in cui diverse funzioni debbano chiamare la stessa funzione. Peggio ancora: cercare di aggiungere o rimuovere le dichiarazioni durante la manutenzione del programma può essere una vera sofferenza. Per queste ragioni, in questo libro le dichiarazioni delle funzioni verranno dichiarate sempre al di fuori del corpo delle altre funzioni.

**D:** Se diverse funzioni hanno lo stesso tipo restituito, le loro dichiarazioni possono essere combinate assieme? Per esempio: dato che sia `print_pun` che `print_count` hanno `void` come tipo restituito, è ammessa la seguente dichiarazione?

```
void print_pun(void), print_count(int n);
```

**R:** Sì, infatti il C ci permette persino di combinare le dichiarazioni delle funzioni con quelle delle variabili:

```
double x, y, average(double a, double b);
```

Nonostante ciò combinare le dichiarazioni in questo modo non è una buona idea perché può creare facilmente confusione.

**D:** Cosa succede se specifichiamo la lunghezza di un parametro costituito da un vettore unidimensionale? [p. 203]

**R:** Il compilatore la ignora. Considerate il seguente esempio:

```
double inner_product(double v[3], double w[3]);
```

A parte documentare che ci si aspetta che gli argomenti di `inner_product` siano dei vettori di lunghezza 3, aver specificato la lunghezza non ha prodotto molto. Il compilatore non controllerà che gli argomenti abbiano davvero una lunghezza pari a 3 e quindi non c'è nessuna sicurezza aggiuntiva. In effetti questa pratica è fuorviante in quanto fa credere che a `inner_product` possano essere passati solo vettori di lunghezza 3, mentre di fatto è possibile passare vettori di una lunghezza qualsiasi.

**D:** Perché si può fare a meno di specificare la prima dimensione di un parametro costituito da un vettore, mentre non è possibile farlo per le altre dimensioni? [p. 205]

**R:** Per prima cosa abbiamo bisogno di discutere su come, nel C, i vettori vengano passati. Come viene spiegato nella Sezione 12.3, quando un vettore viene passato a una funzione, a questa viene dato un *puntatore* al primo elemento del vettore stesso.

Successivamente abbiamo bisogno di sapere come funziona l'operatore di indicizzazione. Supponete che a sia un vettore unidimensionale che viene passato a una funzione. Quando scriviamo

```
a[i] = 0;
```

il compilatore genera una funzione che calcola l'indirizzo di `a[i]` moltiplicando `i` per la dimensione di un elemento del vettore e sommando l'indirizzo rappresentato da `a` al risultato ottenuto. Questo calcolo non dipende dalla lunghezza di `a`, il che spiega perché possiamo ometterla quando definiamo una funzione.

E riguardo ai vettori multidimensionali? Ricordatevi che il C memorizza i vettori ordinandoli per righe, ovvero vengono memorizzati prima gli elementi della riga 0, poi quelli della riga 1 e così via. Supponete che a sia un parametro costituito da un vettore bidimensionale e scrivete

```
a[i][j] = 0;
```

Il compilatore genera delle istruzioni che fanno le seguenti cose: (1) moltiplicare `i` per la dimensione di una singola riga di `a`; (2) sommare al risultato ottenuto l'indirizzo

rappresentato da a; (3) moltiplicare j per la dimensione di un elemento; (4) sommare il risultato ottenuto all'indirizzo calcolato al passo 2. Per generare queste istruzioni il compilatore deve conoscere la dimensione di una riga del vettore che è determinata dal numero delle sue colonne. Di conseguenza il programmatore deve dichiarare il numero di colonne di a.

**D: Perché alcuni programmatori mettono delle parentesi attorno alle espressioni delle istruzioni return?**

R: Gli esempi presenti nella prima edizione del libro di Kernighan e Ritchie avevano sempre delle parentesi nelle istruzioni return, anche se non erano necessarie. I programmatori (e gli autori di libri successivi) hanno preso questa abitudine da K&R. Nel presente volume non useremo queste parentesi dato che non sono necessarie e non danno alcun contributo alla leggibilità (apparentemente Kernighan e Ritchie sono d'accordo: nella seconda edizione del loro libro le istruzioni return non avevano parentesi).

**D: Cosa succede se una funzione non-void cerca di eseguire un'istruzione return priva di espressione? [p. 210]**

R: Questo dipende dalla versione del C in uso. Nel C89 eseguire un return senza espressione all'interno di una funzione non-void causa un comportamento indefinito (ma solo se il programma cerca di utilizzare il valore restituito). Nel C99 questa istruzione è illegale e il compilatore dovrebbe indicarla come un errore.

**D: Come posso controllare il valore restituito dal main per capire se il programma è terminato normalmente? [p. 211]**

R: Questo dipende dal vostro sistema operativo. Molti sistemi operativi permettono che questo valore venga testato all'interno di un "file batch" o all'interno di uno "script di shell" che contiene i comandi per eseguire diversi programmi. Per esempio, in un file batch di Windows la riga

```
if errorlevel 1 commando
```

esegue *commando* se l'ultimo programma è terminato con un codice di stato maggiore o uguale a 1.

In UNIX ogni shell ha un suo metodo per testare il codice di stato. Nella shell Bourne, la variabile \$? contiene lo stato dell'ultimo programma eseguito. La shell C possiede una variabile simile, ma il suo nome è \$status.

**D: Perché durante la compilazione del main il compilatore produce il messaggio di warning "control reaches end of non-void function"?**

R: Il compilatore ha notato che il main non ha un'istruzione return sebbene il suo tipo restituito sia int. Mettere l'istruzione

```
return 0;
```

alla fine del main farà felice il compilatore. Tra l'altro, questa è una buona prassi anche se il vostro compilatore non fa obiezioni sulla mancanza dell'istruzione return.

Quando un programma viene compilato con un compilatore C99, questo warning non si verifica. Nel C99 è ammesso "cadere" fuori dal main senza restituire un valore. Lo standard stabilisce che in questa situazione il main restituisca automaticamente uno 0.

**D:** Riguardo alla domanda precedente: perché non imponiamo semplicemente che il tipo restituito del main sia void?

**R:** Sebbene questa pratica sia piuttosto comune, non è ammessa dallo standard C89. Tuttavia non sarebbe una buona idea nemmeno se fosse ammessa dato che presume che nessuno vada mai a testare lo stato del programma dopo il suo termine.

Il C99 si apre all'uso di questa pratica permettendo che il main venga dichiarato in "qualche altro modo definito dall'implementazione" (quindi con valore restituito diverso da int o con parametri diversi da quelli specificati dallo standard). Tuttavia nessuno di questi utilizzi è portabile e quindi la cosa migliore è dichiarare il valore restituito dal main come int.

**D:** È possibile che la funzione f1 chiami la funzione f2 che a sua volta chiama f1?

**R:** Sì. Questa è solo una forma indiretta di ricorsione nella quale una chiamata di f1 ne porta a un'altra (assicuratevi però che almeno una delle due funzioni f1 ed f2 una possa terminare).

## Esercizi

### Domande 9.1

- La funzione seguente, che calcola l'area di un triangolo, contiene due errori. Trovateli e indicate come risolverli (*Suggerimento*: non ci sono errori nella formula).

```
double triangle_area(double base, height)
double product;
{
 product = base * height;
 return product / 2;
}
```

- Scrivete una funzione check(x, y, n) che restituisca un 1 se x e y sono compresi tra 0 e n - 1 inclusi. La funzione deve restituire 0 negli altri casi. Assumete che x, y ed n siano tutti di tipo int.
- Scrivete una funzione gcd(m, n) che calcoli il massimo comun divisore degli interi m ed n (il Progetto di Programmazione 2 del Capitolo 6 descrive l'algoritmo di Euclide per calcolare il MCD).
- Scrivete la funzione day\_of\_year(month, day, year) che restituisca il giorno dell'anno (un intero compreso tra 1 e 366) specificato dai tre argomenti.
- Scrivete una funzione num\_digits(n) che ritorni il numero di cifre presenti in n (che è un intero positivo). *Suggerimento*: per determinare il numero di cifre nel numero n, dividetelo ripetutamente per 10. Quando n raggiunge lo 0, il numero di divisioni eseguite indica quante cifre aveva n originariamente.
- Scrivete una funzione digit(n, k) che restituisca la k-esima cifra (da destra) di n (un intero positivo). Per esempio: digit(829, 1) restituisce 9, digit(829, 2) restituisce 2 e digit(829, 3) restituisce 8. Se k è maggiore del numero di cifre presenti in n, la funzione deve restituire uno 0.

7. Supponete che la funzione `f` abbia la seguente definizione:

```
int f(int a, int b) { ... }
```

Quali delle seguenti istruzioni sono ammissibili? (Assumete che `i` sia di tipo `int` e che `x` sia di tipo `double`).

- (a) `i = f(83, 12);`
- (b) `x = f(83, 12);`
- (c) `i = f(3.15, 9.28);`
- (d) `x = f(3.15, 9.28);`
- (e) `f(83, 12);`

- Sezione 9.2** 8. Quale dei seguenti prototipi sarebbe ammissibile per una funzione che non restituisce nulla e che ha un solo parametro `double`?

- (a) `void f(double x);`
- (b) `void f(double);`
- (c) `void f();`
- (d) `f(double x);`

- Sezione 9.3** 9. Quale sarà l'output del seguente programma?

```
#include <stdio.h>

void swap(int a, int b);

int main(void)
{
 int i = 1, j = 2;

 swap(i, j);
 printf("i = %d, j = %d\n", i, j);
 return 0;
}

void swap(int a, int b)
{
 int temp = a;
 a = b;
 b = temp;
}
```

10. Scrivete delle funzioni che restituiscano i seguenti valori (assumete che `a` ed `n` siano dei parametri, dove `a` è un vettore di valori `int`, mentre `n` è la lunghezza del vettore).

- (a) il maggiore tra gli elementi di `a`
- (b) la media degli elementi di `a`
- (c) il numero degli elementi di `a` che sono positivi

11. Scrivete la seguente funzione:

```
float compute_GPA(char grades[], int n);
```

il vettore `grades` conterrà voti letterali (A, B, C, D o F, sia maiuscole che minuscole), mentre `n` è la lunghezza del vettore. La funzione deve restituire la media dei voti (assumete A=4, B=3, C=2, D=1, F=0).

12. Scrivete la seguente funzione:

```
double inner_product(double a[], double b[], int n);
la funzione deve restituire a[0] * b[0] + a[1] * b[1] + ... + a[n-1]*b[n-1].
```

13. Scrivete la seguente funzione, che valuta una posizione negli scacchi:

```
int evaluate_position(char board[8][8]);
```

`board` rappresenta una configurazione dei pezzi su una scacchiera, dove le lettere K (*King*), Q (*Queen*), R (*Rook*), B (*Bishop*), N (*Knight*), P (*Pawn*) rappresentano i pezzi bianchi, mentre le lettere k, q, r, b, n e p rappresentano i pezzi neri. La funzione `evaluate_position` deve fare la somma dei valori dei pezzi bianchi (Q=9, R=5, B=3, N=3, P=1) e la somma dei valori dei pezzi neri (somma fatta allo stesso modo). La funzione deve restituire la differenza tra i due numeri. Questo valore deve essere positivo se il giocatore bianco è in vantaggio e negativo se in vantaggio è il giocatore nero.

- Sezione 9.4** 14. La seguente funzione dovrebbe restituire `true` se qualche elemento del vettore `a` è uguale a 0, mentre deve restituire `false` se tutti gli elementi sono diversi da zero. Purtroppo la funzione contiene un errore. Trovatelo e correggetelo:

```
bool has_zero(int a[], int n)
{
 int i;
 for (i = 0; i < n; i++)
 if (a[i] == 0)
 return true;
 else
 return false;
}
```

15. La seguente (piuttosto confusa) funzione cerca il mediano tra tre numeri. Riscrivete la funzione in modo che abbia una sola istruzione di `return`.

```
double median(double x, double y, double z)
{
 if (x <= y)
 if (y <= z) return y;
 else if (x <= z) return z;
 else return x;
 if (z <= y) return y;
 if (x <= z) return x;
 return z;
}
```

**Sezione 9.6** 16. Condensate la funzione fact allo stesso modo in cui abbiamo condensato la funzione power.

W 17. Riscrivete la funzione fact in modo che non sia più ricorsiva.

18. Scrivete una versione ricorsiva della funzione gcd (guardate l'Esercizio 3). Ecco una strategia da usare per calcolare gcd( $m, n$ ): se  $n$  è 0, restituisci  $m$ ; altrimenti chiama ricorsivamente gcd passando  $n$  come primo argomento ed  $m \% n$  come secondo argomento.

W 19. \*Considerate la seguente "funzione del mistero":

```
void pb(int n)
{
 if (n != 0) {
 pb(n / 2);
 putchar('0' + n % 2);
 }
}
```

Tracciate a mano l'esecuzione della funzione. Successivamente scrivete un programma che chiami la funzione passandole un numero immesso dall'utente. Cosa fa la funzione?

## Progetti di programmazione

1. Scrivete un programma che chieda all'utente di immettere una serie di numeri interi (che verranno memorizzati in un vettore) e poi li ordini invocando la funzione selection\_sort. Quando le viene dato un vettore di  $n$  elementi, la selection\_sort deve fare le seguenti cose:

I. cercare l'elemento più grande all'interno del vettore e poi spostarlo nell'ultima posizione del vettore stesso.

II. chiamarsi ricorsivamente per ordinare i primi  $n - 1$  elementi del vettore.

2. Modificate il Progetto di programmazione 5 del Capitolo 5 in modo che utilizza una funzione per il calcolo dell'ammontare dell'imposta sul reddito. Quando lo viene passato l'ammontare di reddito imponibile, la funzione deve restituire il valore dell'imposta dovuta.

3. Modificate il Progetto di programmazione del Capitolo 8 in modo che includa le seguenti funzioni:

```
void generate_random_walk(char walk[10][10]);
void print_array(char walk[10][10]);
```

per prima cosa il main dovrà chiamare la generate\_random\_walk, la quale prima inizializza il vettore in modo che contenga i caratteri '.' e poi sostituisce alcuni di questi con le lettere dalla A alla Z, così come descritto nel progetto originale. Il main poi dovrà invocare la funzione print\_array per stampare il vettore su schermo.

4. Modificate il Progetto di programmazione 16 del Capitolo 8 in modo che utilizzate le seguenti funzioni:

```
void read_word(int counts[26]);
bool equal_array(int counts1[26], int counts2[26]);
```

il `main` dovrà invocare `read_word` due volte: una per ogni parola che l'utente deve immettere. Mentre `read_word` legge la parola, ne usa le lettere per aggiornare il vettore `counts` nel modo descritto nel progetto originale (il `main` dichiara due vettori, uno per ogni parola. Questi vettori vengono usati per tenere traccia di quante siano le occorrenze di ogni lettera all'interno delle parole). Successivamente il `main` dovrà chiamare la funzione `equal_array`, alla quale verranno passati i due vettori. Questa funzione dovrà restituire `true` se gli elementi nei due vettori sono gli stessi (il che indica che le due parole immesse dall'utente sono anagrammi), in caso contrario dovrà restituire `false`.

5. Modificate il Progetto di Programmazione 17 del Capitolo 8 in modo da includere le seguenti funzioni:

```
void create_magic_square(int n, char magic_square[n][n]);
void print_magic_square(int n, char magic_square[n][n]);
```

Dopo aver ottenuto dall'utente il numero  $n$ , il `main` deve chiamare la funzione `create_magic_square` passandole un vettore  $n \times n$  che viene dichiarato dentro il `main` stesso. La funzione riempirà il vettore con i numeri  $1, 2, \dots, n^2$  nel modo descritto nel progetto originale. *Nota:* se il vostro compilatore non supporta i vettori a lunghezza variabile, allora dichiarate il vettore nel `main` in modo che abbia dimensioni  $99 \times 99$  invece che  $n \times n$  e utilizzate i seguenti prototipi al posto di quelli già forniti:

```
void create_magic_square(int n, char magic_square[99][99]);
void print_magic_square(int n, char magic_square[99][99]);
```

6. Scrivete una funzione che calcoli il valore del seguente polinomio:

$$3x^5 + 2x^4 - 5x^3 - x^2 + 7x - 6$$

Scrivete un programma che chieda all'utente di immettere un valore per  $x$  che deve essere passato alla funzione per il calcolo. Alla fine il programma dovrà visualizzare il valore restituito dalla funzione.

7. La funzione `power` della Sezione 9.6 può essere velocizzata calcolando  $x^n$  in un modo diverso. Per prima cosa osservate che se  $n$  è una potenza di 2, allora  $x^n$  può essere calcolato con degli elevamenti al quadrato. Per esempio:  $x^4$  è il quadrato di  $x^2$  e di conseguenza  $x^4$  può essere calcolato con due sole moltiplicazioni invece che tre. Questa tecnica può essere usata anche quando  $n$  non è una potenza di 2. Se  $n$  è pari allora useremo la formula  $x^n = (x^{n/2})^2$ . Se invece  $n$  è dispari allora  $x^n = x \times x^{n-1}$ . Scrivete quindi una funzione ricorsiva che calcoli  $x^n$  (la ricorsione ha termine quando  $n = 0$ , in tal caso la funzione restituisce un 1). Per testare la vostra funzione scrivete un programma che chieda all'utente di immettere dei valori per  $x$  ed  $n$ , chiami la funzione `power` per calcolare  $x^n$  e infine stampi il valore restituito dalla funzione.

8. Scrivete un programma che simuli il gioco *craps* che viene fatto con due dadi. Al primo lancio il giocatore vince se la somma dei dadi è 7 o 11. Il giocatore perde se la somma è 2, 3 oppure 12. Qualsiasi altra uscita viene chiamata il "punto" e il gioco continua. Su tutte le giocate seguenti il giocatore vince se realizza nuovamente il "punto". Perde invece se ottiene un 7. Qualsiasi altro valore viene ignorato e il gioco continua. Alla fine di ogni partita il programma dovrà chiedere all'utente se vuole giocare ancora. Nel caso in cui l'utente risponda diversamente da *y* o *Y*, il programma, prima di terminarsi, dovrà visualizzare il numero di vittorie e di perdite.

```
You rolled: 8
Your point is 8
You rolled: 3
You rolled: 10
You rolled: 8
You win!
```

Play again? *y*

```
You rolled: 6
Your point is 6
You rolled: 5
You rolled: 12
You rolled: 3
You rolled: 7
You lose!
```

Play again? *y*

```
You rolled: 11
You win!
```

Play again? *n*

Wins: 2 Losses: 1

Scrivete il vostro programma in modo che sia costituito da 3 funzioni: *main*, *roll\_dice* e *play\_game*. Questi sono i prototipi per le ultime due funzioni:

```
int roll_dice(void);
bool play_game(void);
```

*roll\_dice* dovrà generare due numeri casuali, ognuno compreso tra 1 e 6 e poi restituire la somma. La funzione *play\_game* invece dovrà giocare una partita di *craps* (ovvero chiamare *roll\_dice* per determinare l'esito di ogni lancio di dati). La funzione dovrà restituire *true* se il giocatore vince, *false* se il giocatore perde. La funzione *play\_game* dovrà anche essere responsabile della visualizzazione dei messaggi che mostrano gli esiti dei vari lanci. Il *main* dovrà chiamare ripetutamente la funzione *play\_game* tenendo traccia del numero di vittorie e del numero di sconfitte. Dovrà anche visualizzare i messaggi "you win" e "you lose". Suggerimento: usate la funzione *rand* per generare i numeri casuali. Guardate il programma *deal.c* nella Sezione 8.2 per avere un esempio di chiamata alla funzione *rand* e alla funzione collegata *srand*.



# 10 Organizzazione del programma

Avendo trattato le funzioni nel Capitolo 9, ora siamo pronti per confrontarci con le diverse questioni che si presentano quando i programmi hanno più di una funzione. Il capitolo inizia con una discussione sulle differenze tra variabili locali (Sezione 10.1) e variabili esterne (Sezione 10.2). La Sezione 10.3 prende in considerazione i blocchi, ovvero istruzioni composte che contengono delle dichiarazioni. La Sezione 10.4 tratta le regole di scope che si applicano ai nomi locali, ai nomi esterni e a quelli dichiarati nei blocchi. La Sezione 10.5, infine, suggerisce un modo per organizzare i prototipi delle funzioni, le definizioni di funzioni, le dichiarazioni delle variabili e le altre componenti di un programma C.

## 10.1 Variabili locali

Una variabile dichiarata nel corpo di una funzione è detta **locale** alla funzione. Nella funzione seguente, `sum` è una variabile locale:

```
int sum_digits(int n)
{
 int sum = 0; /* variabile locale */

 while (n > 0) {
 sum += n % 10;
 n /= 10;
 }

 return sum;
}
```

Per default le variabili locali hanno le seguenti proprietà.

- **Durata della memorizzazione automatica.** La durata della memorizzazione (o estensione) di una variabile è la porzione di esecuzione del programma durante la quale la variabile esiste. Lo spazio per una variabile locale viene allocato "automaticamente" nel momento in cui viene invocata la funzione che la contiene, mentre viene deallocato quando la funzione ha termine. Per questo motivo si dice che le variabili locali hanno una **durata della memorizzazione**.

**automatica.** Una variabile locale non mantiene il suo valore quando la funzione che la contiene ha termine e quindi, quando la funzione viene nuovamente invocata, non c'è alcuna garanzia che la variabile possieda ancora il suo vecchio valore.

- **Scope di blocco.** Lo **scope** della variabile è la porzione del testo di un programma entro la quale si può fare riferimento alla variabile stessa. Una variabile locale ha uno **scope di blocco**: ovvero è visibile dal punto della sua dichiarazione fino alla fine del corpo della funzione che la contiene. Dato che lo scope di una variabile locale non si estende al di fuori della funzione alla quale appartiene, le altre funzioni possono usare il suo nome per altri scopi.

La Sezione 18.2 tratta con maggior dettaglio questo e altri concetti collegati.

Da quando il C99 non richiede che la dichiarazione delle variabili si trovi all'inizio di una funzione, è possibile che una variabile locale abbia uno scope molto piccolo. Nell'esempio seguente, lo scope di i inizia a partire dalla riga nella quale viene dichiarata, la quale può trovarsi vicino alla fine del corpo della funzione:

```
void f(void)
{
 ...
 int i; —— scope of i
 ...
}
```

## Variabili locali statiche

Mettere la parola **static** nella dichiarazione di una variabile locale fa sì che questa abbia una **durata di memorizzazione statica** invece di averne una automatica. Una variabile con una durata di memorizzazione statica possiede una locazione di memoria permanente e quindi può mantenere il suo valore durante l'esecuzione del programma. Considerate la seguente funzione:

```
void f(void)
{
 static int i; /* variabile locale statica */
 ...
}
```

Dato che la variabile i è stata dichiarata statica occupa la stessa locazione di memoria durante tutta l'esecuzione del programma. Quando la funzione f termina, i non perde il suo valore.

Una variabile locale statica ha ancora uno scope di blocco e quindi non è visibile dalle altre funzioni. In breve, una variabile statica è un posto dove nascondere dei dati alle altre funzioni e mantenerli per le chiamate future della stessa funzione.

## Parametri

I parametri hanno le stesse proprietà (durata di memorizzazione automatica e scope di blocco) delle variabili locali. Infatti l'unica vera differenza tra parametri e variabili locali è che ogni parametro viene inizializzato automaticamente quando viene invocata la funzione (gli viene assegnato il valore corrispondente all'argomento).

## 10.2 Variabili esterne

Passare gli argomenti è uno dei modi per trasmettere informazioni a una funzione. Le funzioni possono comunicare anche attraverso le **variabili esterne**, ovvero delle variabili che vengono dichiarate al di fuori del corpo di qualsiasi funzione.

Le proprietà delle variabili esterne (o **variabili globali**, come vengono chiamate a volte) sono diverse da quelle delle variabili locali.

- **Durata della memorizzazione statica.** Le variabili esterne hanno una durata della memorizzazione statica, esattamente come le variabili locali che vengono dichiarate static. Un valore salvato in una variabile esterna vi rimarrà indefinitamente.
- **Scope di file.** Una variabile esterna ha uno **scope di file**: ovvero è visibile a partire dal punto della sua dichiarazione fino alla fine del file che la contiene. Ne risulta che possono avere accesso (e modificare) una variabile esterna tutte le funzioni che seguono la sua dichiarazione.

### Esempio: usare una variabile esterna per implementare uno stack

Per illustrare come possano essere usate le variabili esterne, analizziamo la struttura dati conosciuta come **stack** (lo *stack* o pila è un concetto astratto, non una funzionalità del C, e può essere implementato nella maggior parte dei linguaggi di programmazione). Uno stack, come un vettore, può immagazzinare diversi oggetti dello stesso tipo. Tuttavia le operazioni effettuabili con lo stack sono limitate: possiamo inserire (**push**) un oggetto nello stack (aggiungendolo alla fine cioè sulla "cima dello stack") oppure possiamo prelevare (**pop**) un oggetto dallo stack (rimuovendolo dalla stessa cima). Non è permesso esaminare o modificare un elemento che non si trovi in cima allo stack.

Un modo per implementare uno stack con il C è quello di memorizzare gli oggetti di un vettore che chiameremo **contents**. Una variabile intera chiamata **top** viene usata per indicare la posizione della cima dello stack. Quando lo stack è vuoto, la variabile **top** ha valore 0. Per inserire un oggetto nello stack dobbiamo semplicemente salvarlo in **contents** nella posizione indicata dalla variabile **top** e, successivamente, incrementare il valore di quest'ultima. Eseguire il **pop** di un oggetto richiede che **top** venga prima decrementata e poi usata come indice per caricare da **contents** l'oggetto che deve essere prelevato.

Basato su questo schema, ecco un frammento di programma che dichiara le variabili **contents** e **top** e fornisce un insieme di funzioni che rappresentano le operazioni sullo stack. Tutte e cinque le funzioni devono accedere alla variabile **top**, mentre due delle funzioni necessitano anche dell'accesso al vettore **contents** e quindi rendiamo entrambe le variabili esterne.

```
#include <stdbool.h> /* solo C99 */

#define STACK_SIZE 100

/* variabili esterne */
int contents[STACK_SIZE];
int top = 0;

void make_empty(void)
{
 top = 0;
}

bool is_empty(void)
{
 return top == 0;
}

bool is_full(void)
{
 return top == STACK_SIZE;
}

void push(int i)
{
 if (is_full())
 stack_overflow();
 else
 contents[top++] = i;
}

int pop(void)
{
 if (is_empty())
 stack_underflow();
 else
 return contents[--top];
}
```

## Pregi e difetti delle variabili esterne

Le variabili esterne sono utili quando molte funzioni devono condividere una variabile o quando poche funzioni devono condividere un gran numero di variabili. Nella maggior parte dei casi, tuttavia, è preferibile che le funzioni comunichino attraverso parametri piuttosto che condividendo delle variabili. Ecco perché:

- se modifichiamo una variabile esterna durante la manutenzione del programma (per esempio modificando il suo tipo), dobbiamo controllare quali siano le ripercussioni sulle funzioni che appartengono allo stesso file;

- nel caso in cui a una variabile esterna venisse assegnato un valore non corretto, sarebbe difficile identificare la funzione responsabile. È come tentare di risolvere un omicidio commesso a una festa affollata: non esiste un modo semplice per restringere la lista dei sospetti;
- le funzioni che si basano sulle variabili esterne sono difficili da riutilizzare in altri programmi. Una funzione che dipenda da variabili esterne non è contenuta in sé stessa. Per riutilizzare la funzione dobbiamo trascinarci dietro tutte le variabili esterne di cui ha bisogno.

Molti programmatore C si affidano eccessivamente alle variabili esterne. Uno degli abusi più comuni è quello di utilizzare la stessa variabile esterna per diversi scopi all'interno di funzioni differenti. Supponete che diverse funzioni abbiano bisogno di una variabile *i* per controllare un ciclo *for*. Alcuni programmatore in luogo di dichiarare *i* in ogni funzione che la utilizza, la dichiarano in cima al programma rendendo la variabile visibile a tutte le funzioni. Questa pratica è assolutamente infelice, non solo per le ragioni elencate precedentemente ma anche perché è fuorviante. Qualcuno che leggesse il programma in un secondo momento potrebbe pensare che gli usi della variabile siano collegati quando in realtà non lo sono.

Quando usate delle variabili esterne accertatevi che abbiano dei nomi significativi (le variabili locali non hanno sempre bisogno di nomi significativi: spesso è difficile trovare un nome migliore di *i* per la variabile di controllo di un ciclo *for*). Se vi ritrovate a usare nomi come *i* e *temp* per variabili esterne, allora questo è un sintomo che, probabilmente, queste avrebbero dovuto essere delle variabili locali.



Far diventare esterne variabili che avrebbero dovuto essere locali può condurre a bachi veramente frustranti. Considerate il seguente esempio, dove si suppone che venga visualizzata una disposizione 10 × 10 di asterischi:

```
int i;

void print_one_row(void)
{
 for (i = 1; i <= 10; i++)
 printf("*");
}

void print_all_rows(void)
{
 for (i = 1; i <= 10; i++) {
 print_one_row();
 printf("\n");
 }
}
```

La funzione *print\_all\_rows* stampa solamente una riga invece di 10. Quando la *print\_one\_row* effettua il *return* dopo la sua prima chiamata, *i* ha il valore 11. Successivamente l'istruzione *for* presente in *print\_all\_rows* incrementa *i* e controlla se questa sia minore o uguale a 10. Non è così, di conseguenza il ciclo termina e con lui anche la funzione.

MICHELE HAMMA

## Indovinare un numero

Per acquisire maggiore esperienza con le variabili esterne, scriviamo un semplice programma di gioco. Il programma genera un numero casuale compreso tra 1 e 100 che dovrà essere indovinato dall'utente nel minor numero possibile di tentativi. Ecco quale sarà l'aspetto del programma durante l'esecuzione:

Guess the secret number between 1 and 100.

A new number has been chosen.

Enter guess: 55

Too low; try again.

Enter guess: 65

Too high; try again.

Enter guess: 60

Too high; try again.

Enter guess: 58

You won in 4 guesses!

Play again? (Y/N) y

A new number has been chosen.

Enter guess: 78

Too high; try again.

Enter guess: 34

You won in 2 guesses!

Play again? (Y/N) n

Questo programma dovrà occuparsi di diversi compiti: inizializzare il generatore di numeri casuali, scegliere il numero segreto e interagire con l'utente fino a quando viene scelto il numero corretto. Scrivendo una diversa funzione per ognuno di questi compiti, potremo ottenere il seguente programma:

```
/* Chiede all'utente di indovinare un numero */

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

#define MAX_NUMBER 100

/* variabili esterne */
int secret_number;

/* prototipi */
void initialize_number_generator(void);
void choose_new_secret_number(void);
void read_guesses(void);

int main(void)
{
 char command;
```

```
printf("Guess the secret number between 1 and %d.\n\n",
 MAX_NUMBER);
initialize_number_generator();
do {
 choose_new_secret_number();
 printf("A new number has been chosen.\n");
 read_guesses();
 printf("Play again? (Y/N) ");
 scanf(" %c", &command);
 printf("\n");
} while (command == 'y' || command == 'Y');
return 0;
}

/*****************
 * initialize_number_generator: Inizializza il generatore di numeri casuali *
 * usando l'ora corrente. *
 *****************/
void initialize_number_generator(void)
{
 srand((unsigned) time(NULL));
}

/*****************
 * choose_new_secret_number: Sceglie un numero casuale compreso *
 * tra 1 e MAX_NUMBER e lo salva in secret_number. *
 *****************/
void choose_new_secret_number(void)
{
secret_number = rand() % MAX_NUMBER + 1;
}

/*****************
 * read_guesses: Legge ripetutamente i tentativi fatti dall'utente e lo avvisa *
 * se questi sono maggiori, minori o uguali al numero segreto. *
 * Quando l'utente indovina, stampa il numero totale *
 * dei tentativi effettuati *
 *****************/
void read_guesses(void)
{
 int guess, num_guesses = 0;

 for (;;) {
 num_guesses++;
 printf("Enter guess: ");
 scanf("%d", &guess);
 if (guess == secret_number) {
 printf("You won in %d guesses!\n\n", num_guesses);
 return;
 } else if (guess < secret_number)
}
```

```

 printf("Too low; try again.\n");
 else
 printf("Too high; try again.\n");
 }
}
}

```

Per la generazione del numero casuale, il programma si basa sulle funzioni time, srand e rand che abbiamo visto per la prima volta nel programma deal.c (Sezione 8.2) [funzione time > 26.3][funzione srand > 26.2][ funzione rand > 26.2]. Questa volta stiamo scalando il valore della rand in modo che sia compreso tra 1. e MAX\_NUMBER.

Sebbene il programma guess.c funzioni correttamente, si basa su una variabile esterna. Infatti abbiamo dichiarato la variabile secret\_number come esterna in modo che sia la funzione choose\_new\_secret che la read\_guesses potessero accedervi. Modificando di poco le due funzioni è possibile spostare secret\_number nella funzione main. Modificheremo quindi choose\_secret\_number in modo che restituiscia il nuovo numero e riscriveremo read\_guesses in modo che secret\_number possa esserne passato come un argomento.

Di seguito il nuovo programma con le modifiche indicate in grassetto:

```

guess2.c /* Chiede all'utente di indovinare un numero */

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

#define MAX_NUMBER 100

/* prototypes */
void initialize_number_generator(void);
int new_secret_number(void);
void read_guesses(int secret_number);

int main(void)
{
 char command;
 int secret_number;

 printf("Guess the secret number between 1 and %d.\n\n",
 MAX_NUMBER);
 initialize_number_generator();
 do {
 secret_number = new_secret_number();
 printf("A new number has been chosen.\n");
 read_guesses(secret_number);
 printf("Play again? (Y/N) ");
 scanf(" %c", &command);
 printf("\n");
 } while (command == 'y' || command == 'Y');

 return 0;
}

```

```

* initialize_number_generator: Inizializza il generatore *
* di numeri casuali usando *
* l'ora corrente. *

void initialize_number_generator(void)
{
 srand((unsigned) time(NULL));
}

* new_secret_number: Restituisce un numero causale *
* compreso tra 1 e MAX_NUMBER. *

int new_secret_number(void)
{
 return rand() % MAX_NUMBER + 1;
}

* read_guesses: Legge ripetutamente i tentativi fatti dall'utente e lo avvisa *
* se questi sono maggiori, minori o uguali al numero segreto. *
* Quando l'utente indovina, stampa il numero totale *
* dei tentativi effettuati *

void read_guesses(int secret_number)
{
 int guess, num_guesses = 0;
 for (;;) {
 num_guesses++;
 printf("Enter guess: ");
 scanf("%d", &guess);
 if (guess == secret_number) {
 printf("You won in %d guesses!\n\n", num_guesses);
 return;
 } else if (guess < secret_number)
 printf("Too low; try again.\n");
 else
 printf("Too high; try again.\n");
 }
}
```

## 10.3 Blocchi

Nella Sezione 5.2 abbiamo incontrato delle istruzioni composte della forma  
{ istruzioni }

In realtà il C permette anche la scrittura di istruzioni composte contenenti delle dichiarazioni.

{ dichiarazioni ; istruzioni }

Useremo il termine **blocco** per descrivere delle istruzioni composte di questo tipo. Ecco un esempio di blocco:

```
if (i > j) {
 /* scambia i valori di i e j */
 int temp = i;
 i = j;
 j = temp;
}
```

Per default la durata di memorizzazione di una variabile dichiarata all'interno di un blocco è automatica: lo spazio per la variabile viene allocato quando si entra nel blocco e viene deallocated quando se ne esce. La variabile ha uno scope di blocco, quindi non può essere referenziata al di fuori del blocco stesso. Una variabile appartenente a un blocco può essere definita static in modo da darle una durata statica di memorizzazione.

Il corpo di una funzione è un blocco. I blocchi sono utili anche dentro le funzioni nei casi in cui sono necessarie delle variabili temporanee. Nel nostro ultimo esempio avevamo bisogno di una variabile temporanea in modo da poter scambiare i valori di *i* e *j*. Mettere le variabili temporanee all'interno dei blocchi presenta due vantaggi: (1) evita la confusione all'inizio del corpo delle funzioni a causa delle dichiarazioni di variabili che vengono usate solo per brevi periodi. (2) Riduce il numero di conflitti tra i nomi delle variabili. Tornando al nostro esempio, il nome *temp* può essere benissimo usato in altri punti della funzione (la variabile *temp* è strettamente locale al blocco dove è stata dichiarata).

C99

Il C99 permette di dichiarare le variabili in qualsiasi punto di un blocco, nello stesso modo in cui permette di dichiarare le variabili in qualsiasi punto di una funzione.

## 10.4 Scope

All'interno di un programma C lo stesso identificatore può assumere parecchi significati diversi. Le regole dello scope permettono al programmatore (e al compilatore) di determinare quale sia il significato rilevante in un dato punto del programma.

Ecco qual è la regola più importante per lo scope: quando una dichiarazione all'interno di un blocco dà un nome a un identificatore già visibile (perché ha uno scope di file o perché è stato dichiarato in un blocco che circonda quello attuale), allora la nuova dichiarazione nasconde temporaneamente quella vecchia e l'identificatore assume un nuovo significato.

Considerate l'esempio (in qualche modo estremo) che trovate a pagina seguente, dove l'identificatore *i* possiede quattro significati differenti.

```

int i; /* Declaration 1 */
void f(int i) /* Declaration 2 */
{
 i = 1;
}

void g(void)
{
 int i = 2; /* Declaration 3 */
 if (i > 0) {
 int i; /* Declaration 4 */
 i = 3;
 }
 i = 4;
}

void h(void)
{
 i = 5;
}

```

- Nella Dichiarazione 1, i è una variabile con durata di memorizzazione statica e scope di file.

- Nella Dichiarazione 2, i è un parametro con scope di blocco.

- Nella Dichiarazione 3, i è una variabile automatica con scope di blocco.

- Nella Dichiarazione 4, i è nuovamente automatica e con scope di blocco.

La variabile i viene usata cinque volte. Le regole di scope del C ci permettono di determinare il significato di i in ognuno dei seguenti casi.

- L'assegnamento i = 1 si riferisce al parametro della Dichiarazione 2 e non alla variabile della Dichiarazione 1 in quanto la Dichiarazione 2 nasconde la Dichiarazione 1.
- Il test i > 0 si riferisce alla variabile della Dichiarazione 3 in quanto la Dichiarazione 3 nasconde la Dichiarazione 1 e la Dichiarazione 2 è fuori dallo scope.
- L'assegnamento i = 3 si riferisce alla variabile della Dichiarazione 4, la quale nasconde la Dichiarazione 3.
- L'assegnamento i = 4 si riferisce alla variabile della Dichiarazione 3. Non può riferirsi a quella della Dichiarazione 4 perché questa è fuori scope.
- L'assegnamento i = 5 si riferisce alla variabile della Dichiarazione 1.

## 10.5 Organizzare un programma C

Visti gli elementi principali che costituiscono un programma C, è il momento di sviluppare una strategia per la loro disposizione. Per ora assumeremo che un programma

si trovi sempre all'interno di un solo file. Nel Capitolo 15 vedremo come organizzare programmi suddivisi in numerosi file.

Finora abbiamo visto che un programma può contenere i seguenti componenti:

Direttive del preprocessore come `#include` e `#define`

Definizioni di tipi

Dichiarazioni di variabili esterne

Prototipi di funzioni

Definizioni di funzioni

Il C impone solo poche regole circa l'ordine nel quale questi oggetti debbano essere disposti: una direttiva del preprocessore non ha effetto sino a quando non viene incontrata la riga che la contiene. Il nome di un tipo non può essere utilizzato fino a quando non è stato definito. Una variabile non può essere usata fino a quando non è stata dichiarata. Sebbene il C non sia esigente riguardo le funzioni, è fortemente raccomandabile che ogni funzione venga definita o dichiarata precedentemente alla sua prima chiamata (in ogni caso il C99 lo ritiene un obbligo).

Ci sono diversi sistemi per organizzare un programma in modo tale che queste regole vengano rispettate. Ecco un possibile ordine:

Direttive `#include`

Direttive `#define`

Definizioni di tipo

Dichiarazioni di variabili esterne

Prototipi delle funzioni eccetto il `main`

Definizione del `main`

Definizione delle altre funzioni

Ha senso inserire per prime le direttive `#include` in quanto trasportano informazioni che molto probabilmente saranno necessarie in diversi punti del programma. Le direttive `#define` creano le macro, che vengono solitamente usate in tutto il programma. Porre le definizioni dei tipi prima delle dichiarazioni delle variabili esterne è piuttosto logico dato che le dichiarazioni di queste variabili potrebbero riferirsi ai tipi appena definiti. Dichiarare, come passo successivo, le variabili esterne fa sì che queste siano disponibili in tutte le funzioni che seguono. Dichiarare tutte le funzioni a eccezione del `main`, sconsiglia il problema che si verifica quando una funzione viene chiamata prima che il compilatore abbia visto il prototipo. Questa pratica permette tra l'altro di poter disporre le definizioni delle funzioni in un ordine qualsiasi: per esempio mettendole in ordine alfabetico o raggruppando assieme delle funzioni collegate. Definire il `main` prima delle altre funzioni facilita il lettore nella localizzazione del punto di partenza del programma.

Un ultimo suggerimento: fate precedere ogni definizione di funzione da un commento che fornisca il nome della funzione stessa, ne spieghi lo scopo ed elenchi il significato di ogni suo parametro, descriva il valore restituito (se presente) ed elenchi tutti gli effetti secondari (come le modifiche alle variabili esterne).

C99

## Classificare una mano di poker

Per mostrare come possa essere organizzato un programma C, ne scriveremo uno che sarà leggermente più complesso degli esempi trattati finora. Il programma leggerà e classificherà una mano di poker. Ogni carta della mano deve avere sia un seme (cuori, quadri, fiori o picche) che un valore (due, tre, quattro, cinque, sei, sette, otto, nove, dieci, fante, regina, re o asso). Non ammetteremo l'uso dei jolly e assumeremo di ignorare la scala minima (asso, due, tre, quattro, cinque). Il programma leggerà una mano composta da cinque carte e la classificherà in una delle seguenti categorie (elencate in ordine dalla migliore alla peggiore):

- Scala a colore (*straight flush*, sia scala che colore)
- Poker (*four-of-a-kind*, quattro carte dello stesso seme)
- Full (*full house*, un tris e una coppia)
- Colore (*flush*, cinque carte dello stesso colore)
- Scala (*straight*, cinque carte di valore consecutivo)
- Tris (*three-of-a-kind*, tre carte dello stesso valore)
- Doppia Coppia (*two pairs*, due coppie)
- Coppia (*pair*, due carte dello stesso valore)
- Carta alta (*high card*, qualsiasi altra combinazione)

Se una mano rientra in una o più categorie, il programma dovrà scegliere la migliore.

Per semplificare l'input useremo le seguenti abbreviazioni per indicare i valori e i semi (le lettere potranno essere maiuscole o minuscole):

Valori: 2 3 4 5 6 7 8 9 t j q k a

Semi: c d h s

Nel caso l'utente dovesse immettere una carta non valida o se cercasse di immettere due volte la medesima carta, il programma deve generare un messaggio di errore, ignorare la carta immessa e richiederne un'altra. Immettere il numero 0 al posto di una carta comporterà la chiusura del programma.

Una sessione del programma deve presentarsi in questo modo:

Enter a card: 2s

Enter a card: 5s

Enter a card: 4s

Enter a card: 3s

Enter a card: 6s

Straight flush

Enter a card: 8c

Enter a card: as

Enter a card: 8c

Duplicate card; ignored.

Enter a card: 7c

Enter a card: ad

Enter a card: 3h

Pair

```

Enter a card: 6s
Enter a card: d2
Bad card; ignored.
Enter a card: 2d
Enter a card: 9c
Enter a card: 4h
Enter a card: ts

```

High card

Enter card: 0

Da questa descrizione capiamo che il programma deve svolgere tre compiti:

Leggere una mano di cinque carte.

Analizzare la mano in cerca di coppie, scale e così via.

Stampare la classificazione della mano.

Suddivideremo il programma in tre funzioni (`read_cards`, `analyze_hand` e `print_result`) che eseguano i suddetti compiti. Il `main` non farà nulla se non chiamare queste funzioni all'interno di un ciclo infinito. Le funzioni avranno bisogno di condividere un gran numero di informazioni e per questo le faremo comunicare attraverso delle variabili esterne. La funzione `read_cards` salverà le informazioni riguardanti la mano all'interno di diverse variabili esterne. Successivamente la funzione `analyze_hand` esaminerà queste variabili e salverà quanto trovato all'interno di altre variabili esterne che verranno utilizzate da `print_result`.

Basandosi su questo progetto preliminare possiamo iniziare a delineare la struttura del programma:

```

/* le direttive #include vanno qui */
/* le direttive define vanno qui */
/* le dichiarazioni delle variabili esterne vanno qui */
/* prototipi */
void read_cards(void);
void analyze_hand(void);
void print_result(void);

***** * main: Chiama ripetutamente read_cards, analyze_hand e print_result *
***** */

int main(void)
{
 for (;;) {
 read_cards();
 analyze_hand();
 print_result();
 }
}

```

```

*****+
* read_cards: Salva le carte lette nelle variabili esterne. Esegue il controllo *
* per le carte errate e per quelle duplicate *
*****+
void read_cards(void)
{
}

*****+
* analyze_hand: Determina se la mano contiene una scala, un colore, un poker *
* e/o un tris determina il numero delle coppie e salva *
* il risultato all'interno nelle variabili esterne *
*****+
void analyze_hand(void)
{
}

*****+
* print_result: Notifica all'utente il risultato usando *
* le variabili esterne impostate da analyze_hand *
*****+
void print_result(void)
{
}

```

La questione più urgente rimane quella che riguarda la rappresentazione della mano di gioco. Pensiamo a quali operazioni debbano compiere le funzioni `read_card` e `analyze_hand`. Durante l'analisi della mano, la funzione `analyze_hand` avrà bisogno di conoscere quante carte sono presenti nella mano per ogni seme e per ogni valore. Questo fatto ci suggerisce di utilizzare due vettori: `num_in_rank` e `num_in_suit`. Il valore di `num_in_rank[r]` sarà uguale al numero delle carte di valore `r`, mentre il valore di `num_in_suit[s]` sarà uguale al numero delle carte di seme `s` (codificheremo i valori con i numeri compresi tra 0 e 12 e i semi con i numeri compresi tra 0 e 3). Avremo bisogno di un terzo vettore: `card_exists` che verrà usato da `read_cards` per individuare le carte duplicate. Ogni volta che `read_cards` leggerà una carta di valore `r` e seme `s`, controllerà se il valore di `card_exists[r][s]` è uguale a `true`. In tal caso significherebbe che la carta era già stata immessa. In caso contrario la funzione `read_card` assegnerà il valore `true` all'elemento `card_exists[r][s]`.

Sia la funzione `read_cards` che la funzione `analyze_hand` avranno bisogno di accedere ai vettori `num_in_rank` e `num_in_suit`. Per questo motivo le faremo diventare variabili esterne. Il vettore `card_exists` viene utilizzato solo da `read_cards` e di conseguenza può essere dichiarato come una funzione locale. Di regola le variabili devono essere esterne solo se necessario.

Avendo definito le strutture dati più importanti, possiamo finire il programma:

```

poker.c /* Classifica una mano di poker */

#include <stdbool.h> /* solo C99 */
#include <stdio.h>
#include <stdlib.h>

#define NUM_RANKS 13
#define NUM_SUITS 4
#define NUM_CARDS 5

/* variabili esterne */
int num_in_rank[NUM_RANKS];
int num_in_suit[NUM_SUITS];
bool straight, flush, four, three;
int pairs; /* può essere 0, 1, o 2 */

/* prototipi */
void read_cards(void);
void analyze_hand(void);
void print_result(void);

/*****************
 * main:Chiama ripetutamente read_cards, analyze_hand e print_result *
*****************/
int main(void)
{
for (;;) {
read_cards();
analyze_hand();
print_result();
}
}

/*****************
 * read_cards:Salva le carte lette nelle variabili esterne num_in_rank *
 * e num_in_suit. Esegue il controllo per le carte errate *
 * e per quelle duplicate *
*****************/
void read_cards(void)
{
 bool card_exists[NUM_RANKS][NUM_SUITS];
 char ch, rank_ch, suit_ch;
 int rank, suit;
 bool bad_card;
 int cards_read = 0;

 for (rank = 0; rank < NUM_RANKS; rank++) {
 num_in_rank[rank] = 0;
 for (suit = 0; suit < NUM_SUITS; suit++)
 card_exists[rank][suit] = false;
 }
}

```

```
for (suit = 0; suit < NUM_SUITS; suit++)
 num_in_suit[suit] = 0;

while (cards_read < NUM_CARDS) {
 bad_card = false;

 printf("Enter a card: ");

 rank_ch = getchar();
 switch (rank_ch) {

 case '0': exit(EXIT_SUCCESS);
 case '2': rank = 0; break;
 case '3': rank = 1; break;
 case '4': rank = 2; break;
 case '5': rank = 3; break;
 case '6': rank = 4; break;
 case '7': rank = 5; break;
 case '8': rank = 6; break;
 case '9': rank = 7; break;
 case 't': case 'T': rank = 8; break;
 case 'j': case 'J': rank = 9; break;
 case 'q': case 'Q': rank = 10; break;
 case 'k': case 'K': rank = 11; break;
 case 'a': case 'A': rank = 12; break;
 default: bad_card = true;
 }

 suit_ch = getchar();
 switch (suit_ch) {
 case 'c': case 'C': suit = 0; break;
 case 'd': case 'D': suit = 1; break;
 case 'h': case 'H': suit = 2; break;
 case 's': case 'S': suit = 3; break;
 default: bad_card = true;
 }

 while ((ch = getchar()) != '\n')
 if (ch != ' ') bad_card = true;

 if (bad_card)
 printf("Bad card; ignored.\n");
 else if (card_exists[rank][suit])
 printf("Duplicate card; ignored.\n");
 else {
 num_in_rank[rank]++;
 num_in_suit[suit]++;
 card_exists[rank][suit] = true;
 cards_read++;
 }
}
```

```

*****+
* analyze_hand: Determina se la mano contiene una scala, un colore, *
* un poker e/o un tris; determina il numero delle coppie e salva *
* il risultato all'interno nelle variabili esterne straight, *
* flush, four, three e pairs
*****+
void analyze_hand(void)
{
 int num_consec = 0;
 int rank, suit;
 straight = false;
 flush = false;
 four = false;
 three = false;
 pairs = 0;

 /* controlla se è un colore */
 for (suit = 0; suit < NUM_SUITS; suit++)
 if (num_in_suit[suit] == NUM_CARDS)
 flush = true;

 /* controlla se è una scala */
 rank = 0;
 while (num_in_rank[rank] == 0) rank++;
 for (; rank < NUM_RANKS && num_in_rank[rank] > 0; rank++)
 num_consec++;
 if (num_consec == NUM_CARDS) {
 straight = true;
 return;
 }

 /* fa il controllo per il poker, il tris e le coppie */
 for (rank = 0; rank < NUM_RANKS; rank++) {
 if (num_in_rank[rank] == 4) four = true;
 if (num_in_rank[rank] == 3) three = true;
 if (num_in_rank[rank] == 2) pairs++;
 }
}

*****+
* print_result: Stampa la classificazione della mano basandosi sui valori *
* delle variabili esterne straight, flush, four, three e pairs. *
*****+
void print_result(void)
{
 if (straight && flush) printf("Straight flush");
 else if (four) printf("Four of a kind");
}

```

```

else if (three &&
 pairs == 1) printf("Full house");
else if (flush) printf("Flush");
else if (straight) printf("Straight");
else if (three) printf("Three of a kind");
else if (pairs == 2) printf("Two pairs");
else if (pairs == 1) printf("Pair");
else printf("High card");

printf("\n\n");
}

```

Osservate l'utilizzo della funzione exit all'interno della funzione read\_cards (nel caso '0' del primo costrutto switch). La funzione exit è particolarmente adatta al nostro caso grazie alla sua abilità di terminare l'esecuzione del programma da qualsiasi punto la si invochi.

## Domande & Risposte

**D:** Qual è l'effetto delle variabili locali con durata di memorizzazione statica sulle funzioni ricorsive? [p. 230]

**R:** Quando una funzione viene chiamata ricorsivamente, a ogni invocazione vengono fatte nuove copie delle sue variabili automatiche. Questo però non succede con le variabili statiche. Infatti ogni chiamata alla funzione condividerà la stessa variabile statica.

**D:** Nell'esempio seguente la variabile j viene inizializzata allo stesso valore della variabile i, ma ci sono due variabili chiamate i:

```

int i = 1;

void f(void)
{
 int j = i;
 int i = 2;

}

```

Questo codice è ammissibile? E in tal caso quale sarà il valore iniziale di j, 1 o 2?

**R:** Questo codice è effettivamente ammissibile. Lo scope di una variabile locale non inizia fino a che questa non viene dichiarata. Di conseguenza, la dichiarazione di j si riferisce alla variabile esterna chiamata i. Quindi il valore iniziale di j sarà 1.

## Esercizi

- La seguente bozza di programma mostra solo le definizioni delle funzioni e la dichiarazioni delle variabili.

```

int a;
void f(int b)
{
 int c;
}
void g(void)
{
 int d;
 {
 int e;
 }
}
int main(void)
{
 int f;
}

```

Per ognuno dei seguenti scope, elencate tutte le variabili e i nomi dei parametri visibili nello scope stesso:

- (a) La funzione *f*
  - (b) La funzione *g*
  - (c) Il blocco dove viene dichiarata *e*
  - (d) La funzione *main*
2. La seguente bozza di programma illustra solo la definizioni delle funzioni e le dichiarazioni delle variabili.

```

int b, c,
void f(void)
{
 int b, d;
}
void g(int a)
{
 int c;
 {
 int a, d;
 }
}
int main(void)
{
 int c, d;
}

```

Per ognuno dei seguenti scope, elencate tutte le variabili e i nomi dei parametri visibili nello scope stesso. Se è presente più di una variabile o parametro con lo stesso nome, indicate quale di questi è visibile.

- (e) la funzione `f`
  - (f) la funzione `g`
  - (g) il blocco dove vengono dichiarate `a` e `d`
  - (h) la funzione `main`
3. \*Supponete che un programma abbia un'unica funzione (`main`). Quante variabili chiamate i può contenere un programma di questo tipo?

## Progetti di programmazione

1. Modificate l'esempio dello stack della Sezione 10.2 in modo che memorizzi caratteri invece di interi. Successivamente aggiungete una funzione `main` che chieda all'utente di immettere una serie di parentesi tonde e/o graffe. Il programma dovrà indicare se le parentesi sono annidate in modo appropriato o meno:

Enter parentheses and/or braces: `(({}{}){})`  
Parentheses/braces are nested properly

*Suggerimento:* quando il programma legge un carattere, fate in modo che immetta nello stack ogni parentesi aperta (sia tonda che graffa). Quando il programma legge una parentesi chiusa deve eseguire un'operazione di pop dallo stack e controllare che l'oggetto estratto sia la parentesi corrispondente (altrimenti vorrebbe dire che le parentesi non sono state annidate correttamente). Quando il programma legge il carattere new-line, deve controllare lo stato dello stack. Nel caso in cui questo fosse vuoto significherebbe che le parentesi erano tutte abbinate, altrimenti, se lo stack *non* fosse vuoto (o se venisse chiamata la funzione `stack_underflow`) vorrebbe dire che le parentesi non erano abbinate a dovere. Se la funzione `stack_overflow` viene chiamata, il programma deve stampare il messaggio `Stack overflow` e chiudersi immediatamente.

2. Modificate il programma `poker.c` della Sezione 10.5 spostando all'interno del `main` i vettori `num_in_rank` e `num_in_suit`. La funzione `main` passerà questi argomenti alle funzioni `read_cards` e `analyze_cards`.
3. Rimuovete dal programma `poker.c` della Sezione 10.5 i vettori `num_in_rank`, `num_in_suit` e `card_exists`. Al loro posto fate in modo che il programma memorizzi le carte in un vettore  $5 \times 2$ . Ogni riga del vettore dovrà rappresentare una carta. Ad esempio: se il vettore viene chiamato `hand`, allora `hand[0][0]` conterrà il valore della prima carta mentre `hand[0][1]` conterrà il seme della prima carta.
4. Modificate il programma `poker.c` della Sezione 10.5 in modo che riconosca una categoria addizionale: il "royal flush" (la scala reale costituita da un asso, un re, una regina, un fante e un dieci dello stesso seme). Un royal flush ha valore più alto di tutte le altre combinazioni.
5. Modificate il programma `poker.c` della Sezione 10.5 in modo da ammettere le scale minime (asso, due, tre, quattro, cinque).
6. Alcune calcolatrici (in modo particolare quelle della Hewlett-Packard) utilizzano un sistema per la scrittura delle espressioni matematiche conosciuto come *Reverse Polish Notation* (RPN). Scrivete un programma che converta un'espressione matematica scritta in notazione prefissa (ad esempio `+ 3 4` per  $3 + 4$ ) in notazione RPN. Il programma dovrà quindi calcolare il risultato dell'espressione.

*Polish Notation (RPN).* In questa notazione gli operatori non vengono posti *tra* gli operandi bensì *dopo* questi ultimi. Per esempio: in RPN  $1 + 2$  si scriverebbe  $1\ 2\ +$ , mentre  $1 + 2 * 3$  verrebbe scritto come  $1\ 2\ 3\ *\ +$ . Le espressioni RPN possono essere calcolate facilmente facendo uso di uno stack. L'algoritmo coinvolge la lettura degli operatori e degli operandi presenti all'interno di un'espressione seguendo un ordine che va da sinistra a destra, nonché le seguenti operazioni:

quando si incontra un operando, questo deve essere immesso nello stack

quando si incontra un operatore, occorre: prelevare i suoi operandi dallo stack, eseguire l'operazione su questi operandi e poi inserire il risultato nello stack.

Scrivete un programma che calcoli le espressioni RPN. Gli operandi saranno degli interi costituiti da una singola cifra. Gli operatori sono:  $+, -, *, /$  e  $=$ . L'operatore  $=$  fa sì che venga visualizzato l'elemento presente nella cima dello stack, che lo stack stesso venga svuotato e che un'altra espressione venga richiesta all'utente. Il processo continua fino a quando l'utente non immette un carattere che non è un operatore o un operando:

Enter an RPN expression: 1 2 3 \* + =

Value of expression: 7

Enter an RPN expression: 5 8 \* 4 9 - / =

Value of expression: -8

Enter an RPN expression: g

Se lo stack va in overflow, il programma dovrà stampare il messaggio *Expression is too complex* e poi chiudersi. Se lo stack va in underflow (a causa di un'espressione come  $1\ 2\ + +$ ), il programma dovrà visualizzare il messaggio *Not enough operands in expression* e chiudersi. Suggerimento: nel vostro programma utilizzate il codice dello stack della Sezione 10.2. Per leggere gli operandi e gli operatori usate l'istruzione `scanf(" %c", &ch)`.

7. Scrivete un programma che chieda all'utente di immettere un numero e successivamente visualizzi quel numero utilizzando dei caratteri per simulare l'effetto di un display a sette segmenti:

Enter a number: 491-9014



I caratteri diversi dalle cifre devono essere ignorati. Scrivete il programma in modo che il massimo numero di cifre sia controllato dalla macro `MAX_DIGITS`, la quale deve avere un valore pari a 10. Se il numero da visualizzare contiene un numero maggiore di cifre, le cifre eccedenti devono essere ignorate. Suggerimento: usate due vettori esterni. Uno sarà il `segment_array` (vedere l'Esercizio 6 del Capitolo 8) che serve per memorizzare i dati rappresentanti la corrispondenza tra cifre e segmenti. L'altro sarà il vettore `digits`: un vettore con 4 righe (dato che ogni cifra scritta con segmenti è alta quattro caratteri) e `MAX_DIGITS * 4` colonne (le cifre sono larghe tre caratteri, ma è necessario uno spazio tra esse per la leg-

gibilità). Scrivete il programma con quattro funzioni: `main`, `clear_digits_array`, `process_digit` e `print_digits_array`. Ecco i prototipi delle funzioni:

```
void clear_digits_array(void);
void process_digit(int digit, int position);
void print_digits_array(void);
```

`clear_digits_array` memorizzerà caratteri vuoti in tutti gli elementi del vettore `digits`. La funzione `process_digit` salverà la rappresentazione a sette segmenti di digit all'interno in una specifica posizione del vettore `digits` (le posizioni andranno da 0 a `MAX_DIGITS - 1`). La funzione `print_digits_array` visualizzerà le righe del vettore `digits`, ognuna su una riga a sé stante, producendo un output simile a quello mostrato nell'esempio.



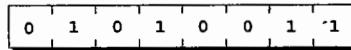
# 11 Puntatori

I puntatori sono una delle caratteristiche più importanti (e spesso meno comprese) del C. In questo capitolo ci concentreremo sugli aspetti base, mentre nel Capitolo 12 e nel Capitolo 17 tratteremo gli usi più avanzati dei puntatori.

Inizieremo con una discussione sugli indirizzi di memoria e sulla relazione che questi hanno con le variabili puntatore (Sezione 11.1). Successivamente la Sezione 11.2 introdurrà l'operatore di indirizzo e l'operatore asterisco. La Sezione 11.3 tratterà l'assegnamento dei puntatori. La Sezione 11.4 spiegherà come passare dei puntatori a funzione, mentre la Sezione 11.5 parlerà della restituzione dei puntatori da parte delle funzioni.

## 11.1 Variabili puntatore

Il primo passo per capire i puntatori è visualizzare cosa rappresentino a livello macchina. Nella maggior parte dei computer moderni la memoria è suddivisa in **byte**, ognuno dei quali è in grado di memorizzare otto bit di informazione:

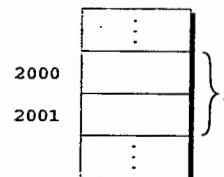


Ogni byte possiede un **indirizzo** univoco che lo distingue dagli altri presenti in memoria. Se nella memoria ci sono  $n$  byte, allora possiamo pensare che gli indirizzi vadano da 0 a  $n - 1$  (guardate la figura a pagina seguente).

Un programma eseguibile è costituito sia dal codice (istruzioni macchina corrispondenti ai costrutti del programma C originale) che dai dati (variabili del programma originale). Ogni variabile presente nel programma occupa uno o più byte della memoria. L'indirizzo del suo primo byte viene considerato l'indirizzo della variabile stessa.

| Indirizzo | Contenuto |
|-----------|-----------|
| 0         | 01010011  |
| 1         | 01110101  |
| 2         | 01110011  |
| 3         | 01100001  |
| 4         | 01101110  |
| :         | :         |
| n-1       | 01000011  |

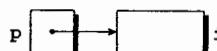
Nella figura seguente la variabile i occupa i byte corrispondenti agli indirizzi 2000 e 2001, di conseguenza l'indirizzo di i è 2000:



È qui che entrano in gioco i puntatori. Sebbene gli indirizzi siano rappresentati da numeri, il loro intervallo di valori può differire da quello degli interi, di conseguenza non possiamo salvarli nelle variabili intere ordinarie. Possiamo invece memorizzarli all'interno di speciali variabili: le **variabili puntatore**. Quando memorizziamo l'indirizzo di una variabile i in una variabile puntatore p diciamo che p "punta" a i. In altre parole: un puntatore non è altro che un indirizzo, e una variabile puntatore è semplicemente una variabile che può memorizzare quell'indirizzo.

D&amp;R

Nei nostri esempi, invece di mostrare i puntatori come degli indirizzi, faremo uso di una notazione più semplice. Per indicare che una variabile puntatore p contiene l'indirizzo della variabile i, illustreremo graficamente il contenuto di p come una freccia che si dirige verso i:



## Dichiarare una variabile puntatore

Una variabile puntatore viene dichiarata praticamente allo stesso modo in cui viene dichiarata una variabile normale. L'unica differenza è che il nome di una variabile puntatore deve essere preceduto da un asterisco:

```
int *p;
```

Questa dichiarazione stabilisce che p è una variabile puntatore in grado di puntare a oggetti di tipo int. Usiamo il termine oggetto invece di *variabile* dato che, come

vedremo nel Capitolo 17, p può puntare a un'area di memoria che non appartiene a una variabile (fate attenzione al fatto che il termine "oggetto" avrà un significato diverso quando nel Capitolo 19 discuteremo della progettazione di un programma [oggetti astratti > 19.1]).

Le variabili puntatore possono comparire nelle dichiarazioni assieme ad altre variabili:

```
int i, j, a[10], b[20], *p, *q;
```

In questo esempio sia i che j sono delle normali variabili intere, a e b sono vettori di interi, mentre p e q sono puntatori a oggetti di tipo intero.

Il C richiede che ogni variabile puntatore punti solamente a oggetti di un particolare tipo (il **tipo del riferimento**):

```
int *p; /* punta solo a interi */
double *q; /* punta solo a double */
char *r; /* punta solo a caratteri */
```

non ci sono restrizioni su quale possa essere il tipo riferito. In effetti una variabile puntatore può persino puntare a un altro puntatore [**puntatori a puntatori > 17.6**].

## 11.2 L'operatore indirizzo e l'operatore asterisco

Il C fornisce una coppia di operatori che sono specificatamente pensati per l'utilizzo con i puntatori. Per trovare l'indirizzo di una variabile useremo l'operatore & (indirizzo). Se x è una variabile, allora &x è il suo indirizzo di memoria. Per guadagnare accesso all'oggetto puntato da un puntatore useremo l'operatore \* (chiamato anche operatore **indirection**). Se p è un puntatore allora \*p rappresenta l'oggetto al quale p sta puntando.

### L'operatore indirizzo

Dichiarare una variabile puntatore prepara lo spazio per un puntatore ma non la fa puntare ad alcun oggetto:

```
int *p; /* non punta ad alcun oggetto in particolare */
```

Provvedere all'inizializzazione della variabile p prima di utilizzarla è essenziale. Un modo per inizializzare una variabile puntatore è quello di assegnarle l'indirizzo di qualche variabile (o più genericamente un lvalue [**lvalue > 4.2**]) utilizzando l'operatore &:

```
int i, *p;
-
p = &i;
```

Questa istruzione, assegnando l'indirizzo di i alla variabile p, fa sì che p punti a i.



È possibile anche inizializzare un puntatore nel momento in cui questo dichiarato:

D&R

```
int i;
int *p = &i;
```

Possiamo persino combinare assieme la dichiarazione di *i* con la dichiarazione ammesso però che *i* venga dichiarata per prima:

```
int i, *p = &i;
```

## L'operatore asterisco

Una volta che una variabile puntatore punta a un oggetto, possiamo usare l'operatore \* per accedere a quello che è il contenuto dell'oggetto stesso. Per esempio: se *p* punta a *i*, possiamo stampare il valore di *i* in questo modo:

```
printf("%d\n", *p);
```

D&R

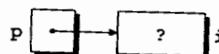
la funzione printf stamperà il *valore* di *i* e non il suo *indirizzo*.

Un lettore portato per la matematica potrebbe pensare che l'operatore \* sia contrario all'operatore &. Applicando un & a una variabile si ottiene un puntatore alla stessa variabile, applicando un \* al puntatore ci riporta alla variabile originale:

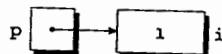
```
j = *&i; /* equivalente a j = i; */
```

Fino a quando *p* punta a *i*, *\*p* è un alias per *i*. Questo significa che *\*p* non solo mantiene lo stesso valore di *i*, ma che cambiando il valore di *\*p* si modifica anche il valore di *i* (*\*p* è un lvalue e quindi è possibile farlo oggetto di assegnamenti). L'esempio seguente illustra l'equivalenza di *\*p* e *i*, le immagini mostrano i valori di *p* e *i* ai vari punti.

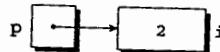
```
p = &i;
```



```
i = 1;
```



```
printf("%d\n", i); /* stampa 1 */
printf("%d\n", *p); /* stampa 1 */
*p = 2;
```



```
printf("%d\n", i); /* stampa 2 */
printf("%d\n", *p); /* stampa 2 */
```



Non applicate mai l'operatore asterisco su una variabile puntatore non inizializzata. Se la variabile puntatore p non è stata inizializzata, qualsiasi tentativo di utilizzare il valore di p provoca un comportamento indefinito. Nell'esempio seguente la chiamata alla printf può stampare cose prive di senso, causare il crash del programma o avere altri effetti indesiderati:

```
int *p;
printf("%d", *p); *** SBAGLIATO ***
```

Assegnare un valore a \*p è particolarmente pericoloso. Se per caso p contiene un indirizzo valido di memoria, il seguente assegnamento cercherà di modificare i dati contenuti in quell'indirizzo:

```
int *p;
*p = 1; *** SBAGLIATO ***
```

Se la locazione modificata da questo assegnamento appartiene al programma, quest'ultimo potrebbe comportarsi in modo imprevedibile. Se invece la localizzazione appartiene al sistema operativo molto probabilmente il programma andrà in crash. Il vostro compilatore potrebbe emettere un messaggio di warning per segnalare che la variabile p non è inizializzata. Fate attenzione quindi ai messaggi di warning che ricevete.

## 11.3 Assegnamento dei puntatori

Il C permette di utilizzare l'operatore di assegnamento per copiare i puntatori, ammesso che questi siano dello stesso tipo. Supponete che i, j, p e q vengano dichiarate in questo modo:

```
int i, j, *p, *q;
```

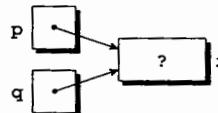
L'istruzione

```
p = &i;
```

è un esempio di assegnamento di un puntatore. L'indirizzo di i viene copiato dentro p. Ecco un altro esempio di assegnamento di un puntatore:

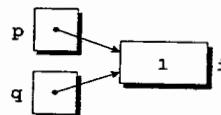
```
q = p;
```

Questa istruzione copia il contenuto di p (l'indirizzo di i) all'interno di q, il che fa sì che q punti allo stesso posto di p.

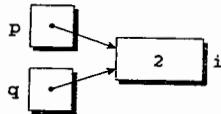


Ora sia p che q puntano a i e quindi possiamo modificare i assegnando un nuovo valore sia a \*p che a \*q:

```
*p = 1;
```



$*q = 2;$



Allo stesso oggetto possono puntare un numero qualsiasi di variabili puntatore.  
Fate attenzione a non confondere

$q = p;$

con

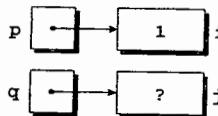
$*q = *p;$

La prima istruzione è l'assegnamento di un puntatore, mentre la seconda, come dimostrano gli esempi seguenti, non lo è affatto:

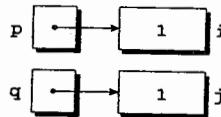
$p = &i;$

$q = &j;$

$i = 1;$



$*q = *p;$



L'assegnamento  $*q = *p$  copia il valore al quale punta p (il valore di i) nell'oggetto puntato da q (la variabile j).

## 11.4 Puntatori usati come argomenti

Fino a questo momento abbiamo evitato una domanda piuttosto importante: a che cosa servono i puntatori? Non esiste un'unica risposta perché nel C i puntatori hanno parecchi utilizzi distinti. In questa sezione vedremo come una variabile puntatore può essere utile se usata come argomento di una funzione. Discuteremo di altri usi dei puntatori nella Sezioni 11.5 e nei Capitoli 12 e 17.

Nella Sezione 9.3 abbiamo visto che una variabile passata come argomento nella chiamata di una funzione viene protetta da ogni modifica perché il C passa gli argomenti per valore. Questa proprietà del C può essere una seccatura se vogliamo che una funzione abbia la possibilità di modificare la variabile. Nella Sezione 9.3 abbiamo provato (e abbiamo fallito) a scrivere una versione della funzione decompose che potesse modificare due dei suoi argomenti.

I puntatori forniscono una soluzione al problema: invece di passare la variabile *x* come argomento della funzione, passeremo *&x*, ovvero un puntatore a *x*. Dichereremo come puntatore il parametro corrispondente *p*. Quando la funzione verrà invocata, *p* avrà il valore *&x* e quindi *\*p* (l'oggetto al quale punta *p*) sarà un alias per *x*. Questo permetterà alla funzione sia di leggere che modificare *x*.

Per vedere in azione questa tecnica modifichiamo la funzione decompose dichiarando come puntatori i parametri *int\_part* e *frac\_part*. Ora la definizione di decompose si presenta in questo modo:

```
void decompose(double x, long *int_part, double *frac_part)
{
 *int_part = (long) x;
 *frac_part = x - *int_part;
}
```

Il prototipo per decompose può essere

```
void decompose(double x, long *int_part, double *frac_part);
```

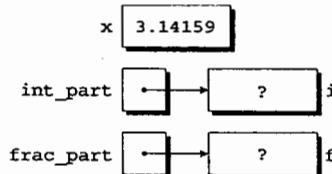
oppure

```
void decompose(double, long *, double *);
```

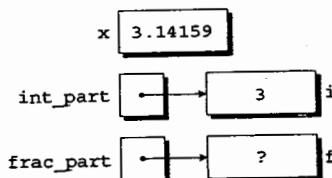
Invocheremo la funzione decompose in questo modo:

```
decompose(3.14159, &i, &d);
```

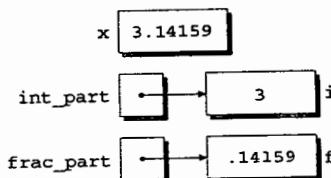
A causa del fatto che l'operatore *&* è stato posto davanti a *i* e *d*, gli argomenti della funzione decompose sono *puntatori* a *i* e a *d*, e non i *valori* di *i* e *d*. Quando la funzione decompose viene chiamata, il valore 3.14159 viene copiato dentro *x*, un puntatore a *i* viene memorizzato all'interno *int\_part* e un puntatore a *d* viene memorizzato all'interno di *frac\_part*:



Il primo assegnamento nel corpo della funzione decompose converte il valore di *x* al tipo *long* e lo salva all'interno dell'oggetto puntato da *int\_part*. Visto che *int\_part* punta a *i*, questo assegnamento mette dentro *i* il valore 3.



Il secondo assegnamento carica il valore puntato da `int_part` (il valore di `i`) che è 3. Questo valore viene convertito al tipo `double` e sottratto a `x` fornendo come risultato 0.14159, il quale viene a sua volta memorizzato nell'oggetto puntato da `frac_part`:



Quando la funzione decompose termina, `i` e `f` avranno rispettivamente i valori 3 e 0.14159. Quindi abbiamo ottenuto quello che volevamo originariamente.

In effetti usare i puntatori come argomento per le funzioni non è nulla di nuovo. Lo stiamo facendo sin dal Capitolo 2 con le chiamate alla funzione `scanf`. Considerate il seguente esempio:

```

int i;
...
scanf("%d", &i);

```

Dobbiamo mettere l'operatore `&` davanti alla variabile `i` in modo che alla `scanf` venga passato un puntatore. Questo indica alla `scanf` dove posizionare il valore letto. Senza l'operatore `&`, alla `scanf` verrebbe passato il valore di `i` invece che il suo indirizzo. Nell'esempio seguente alla `scanf` viene passata una variabile puntatore:

```

int i, *p;
...
p = &i;
scanf("%d", p);

```

Visto che `p` contiene l'indirizzo della variabile `i`, la `scanf` leggerà un intero e lo salverà all'interno della variabile `i`. Utilizzare l'operatore `&` nella chiamata sarebbe stato errato:

```
scanf("%d", &p); /** SBAGLIATO **/
```

in questo caso la `scanf` leggerebbe un intero e lo memorizzerebbe dentro `p` invece che dentro `i`.



Non passare un puntatore a una funzione che ne attende uno può avere conseguenze disastrose. Supponete di chiamare la funzione decompose senza mettere l'operatore & davanti alle variabili i e d:

```
decompose(3.14159, i, d);
```

la funzione decompose si aspetta dei puntatori per il suo secondo e terzo argomento, ma al loro posto le vengono passati i *valori* delle variabili i e d. La funzione non ha modo di riconoscere la differenza e quindi userà quei valori come fossero dei veri puntatori. Quando decompose dovrà memorizzare dei valori in \*int\_part e \*frac\_part, di fatto, invece di modificare i e d, andrà ad agire su locazioni di memoria sconosciute.

Se abbiamo fornito un prototipo per la funzione (come dovremmo sempre fare), allora il compilatore ci farà sapere che stiamo tentando di passare degli argomenti di un tipo non corretto. Il caso della scanf però è diverso, spesso il compilatore non rileva il mancato passaggio di un puntatore e questo rende la funzione particolarmente soggetta agli errori.

## PROGRAMMA

## Trovare il massimo e il minimo in un vettore

Per illustrare come i puntatori vengano passati alle funzioni, diamo un'occhiata consideriamo la funzione chiamata `max_min` che cerca l'elemento più grande e quello più piccolo tra quelli presenti in un vettore. In una chiamata alla `max_min` le passeremo dei puntatori a due variabili in modo che la funzione possa salvare i suoi risultati all'interno di queste ultime. La funzione ha il seguente prototipo:

```
void max_min(int a[], int n, int *max, int *min);
```

Una chiamata alla `max_min` può avere il seguente aspetto:

```
max_min(b, N, &big, &small);
```

dove b è un vettore di interi, N è il numero di elementi di b, big e small sono delle normali variabili intere. Quando `max_min` trova l'elemento più grande presente in b, lo salva nella variabile big grazie a un assegnamento a `*max` (`max` punta a `big` e quindi un assegnamento a `*max` modifica il valore di `big`). Allo stesso modo `max_min` salva il valore del più piccolo elemento di b all'interno della variabile small per mezzo di un assegnamento a `*min`.

Per testare il funzionamento di `max_min`, scriveremo un programma che: legga 10 numeri mettendoli in un vettore, passi quest'ultimo alla funzione `max_min` e stampi il risultato:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
```

```
Largest: 102
```

```
Smallest: 7
```

A pagina seguente un programma completo.

```

maxmin.c /* Cerca il massimo e il minimo in un vettore */

#include <stdio.h>

#define N 10

void max_min(int a[], int n, int *max, int *min);

int main(void)
{
 int b[N], i, big, small;

 printf("Enter %d numbers: ", N);
 for (i = 0; i < N; i++)
 scanf("%d", &b[i]);

 max_min(b, N, &big, &small);

 printf("Largest: %d\n", big);
 printf("Smallest: %d\n", small);

 return 0;
}

void max_min(int a[], int n, int *max, int *min)
{
 int i;

 *max = *min = a[0];
 for (i = 1; i < n; i++) {
 if (a[i] > *max)
 *max = a[i];
 else if (a[i] < *min)
 *min = a[i];
 }
}

```

## Usare const per proteggere gli argomenti

Quando invochiamo una funzione e le passiamo un puntatore a una variabile, di solito assumiamo che la funzione modificherà la variabile (altrimenti perché la funzione dovrebbe richiedere un puntatore?). Per esempio, se in un programma vediamo un'istruzione come questa:

```
f(&x);
```

ci aspettiamo che f modifichi il valore di x. Tuttavia è possibile che f abbia solamente la necessità di esaminare il valore di x ma non quella di modificarlo. La ragione dell'uso di un puntatore può essere l'efficienza: passare il valore di una variabile può essere uno spreco di tempo e spazio se la variabile necessita una quantità di memoria considerevole (la Sezione 12.3 approfondisce questo argomento).

**D&R**

Possiamo usare la const per documentare che una funzione non modificherà un oggetto del quale le viene passato un indirizzo. La parola const deve essere messa nella dichiarazione del parametro, prima di specificare il suo tipo:

```
void f(const int *p)
{
 p = 0; / SBAGLIATO */
}
```

Quest'uso di const indica che p è un puntatore a un "intero costante". Cercare di modificare \*p è un errore che verrà rilevato dal compilatore.

## 11.5 Puntatori usati come valori restituiti

Non solo possiamo passare puntatori a funzioni, ma possiamo anche scrivere funzioni che restituiscano puntatori. Questo tipo di funzioni è relativamente comune, ne incontreremo diverse nel Capitolo 13.

La funzione seguente, dati i puntatori a due interi, restituisce un puntatore al maggiore dei due:

```
int *max(int *a, int *b)
{
 if (*a > *b)
 return a;
 else
 return b;
}
```

Quando invochiamo la funzione max, le passiamo due puntatori a variabili int e salviamo il risultato in una variabile puntatore:

```
int *p, i, j;
-
p = max(&i, &j);
```

Durante la chiamata a max, \*a è un alias per i, mentre \*b è un alias per j. Se i ha un valore maggiore di j, max restituisce l'indirizzo di i, altrimenti restituisce l'indirizzo di j. Dopo la chiamata, p punterà a i oppure a j.

La funzione max restituisce uno dei puntatori che le vengono passati come argomento, tuttavia questa non è l'unica possibilità. Una funzione può anche restituire un puntatore a una variabile esterna oppure a una variabile interna che sia stata dichiarata static.



Non restituite mai un puntatore a una variabile locale *automatica*:

```
int *f(void)
{
 int i;
 -
 return &i;
}
```

La variabile *i* non esiste dopo che la *f* ha avuto termine, di conseguenza il puntatore non sarà valido. In questa situazione alcuni compilatori generano un messaggio di warning come "function returns address of local variable".

I puntatori possono puntare a elementi di un vettore e non solo alle normali variabili. Se *a* è un vettore, allora *&a[i]* è il puntatore all'elemento *i* di *a*. A volte, quando una funzione ha un argomento costituito da un vettore, può essere utile che la funzione restituisca un puntatore a uno degli elementi presenti nel vettore. Per esempio la seguente funzione, assumendo che *a* abbia *n* elementi, restituisce un puntatore all'elemento che si trova nel mezzo di *a*:

```
int *find_middle(int a[], int n) {
 return &a[n/2];
}
```

Il Capitolo 12 esamina nel dettaglio la relazione presente tra i puntatori e i vettori.

## Domande & Risposte

**\*D: Un puntatore corrisponde sempre a un indirizzo? [p. 254]**

**R:** Di solito, ma non sempre. Considerate un computer la cui memoria principale è suddivisa in **word** invece che in byte. Una word può contenere 36, 60 o un qualsiasi altro numero di bit. Ipotizzando word di 36 bit, la memoria si presenterà in questo modo:

| Indirizzo   | Contenuto                            |
|-------------|--------------------------------------|
| 0           | 001010011001010011001010011001010011 |
| 1           | 001110101001110101001110101001110101 |
| 2           | 001110011001110011001110011001110011 |
| 3           | 001100001001100001001100001001100001 |
| 4           | 001101110001101110001101110001101110 |
|             | ⋮                                    |
| <i>n</i> -1 | 001000011001000011001000011001000011 |

Quando la memoria viene divisa in word, ognuna di queste ha un indirizzo. Un intero solitamente occupa una word, di conseguenza un puntatore a un intero può essere un indirizzo. Tuttavia una word può memorizzare più di un carattere. Per esempio: una word a 36 bit può contenere sei caratteri a 6 bit:

|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
| 010011 | 110101 | 110011 | 100001 | 101110 | 000011 |
|--------|--------|--------|--------|--------|--------|

oppure quattro caratteri da 9 bit:

|           |           |           |           |
|-----------|-----------|-----------|-----------|
| 001010011 | 001110101 | 001110011 | 001100001 |
|-----------|-----------|-----------|-----------|

Per questa ragione il puntatore a un carattere deve essere memorizzato in modo diverso rispetto agli altri puntatori. Un puntatore a un carattere può essere costituito da un indirizzo (la word nella quale è contenuto il carattere) più un piccolo intero (la posizione del carattere all'interno della word).

Su alcuni computer i puntatori possono essere degli "offset" e non indirizzi completi. Per esempio: le CPU della famiglia x86 dell'Intel (utilizzata in molti personal computer) possono eseguire programmi secondo diverse modalità. La più vecchia di queste, che risale al processore 8086 del 1978, viene chiamata **real mode**. In questa modalità gli indirizzi sono rappresentati a volte da un singolo numero a 16 bit (un offset) e a volte come una coppia di due numeri a 16 bit (una coppia segmento:offset). Un offset non è un vero indirizzo di memoria, infatti la CPU deve combinarlo con il valore del segmento, che è memorizzato in uno speciale registro. Al fine di supportare il real mode di solito i vecchi compilatori C fornivano due tipi di puntatori: i **near pointer** (offset di 16 bit) e i **far pointer** (coppie segmento: offset di 32 bit). Questi compilatori solitamente riservavano le parole near e far come keyword non standard che potevano essere usate per dichiarare le variabili puntatore.

**D:** Se un puntatore può puntare ai dati in un programma, è possibile anche avere dei puntatori che puntano al codice del programma?

**R:** Sì. Tratteremo i puntatori alle funzioni nella Sezione 17.7.

**D:** Sembra che ci sia un'inconsistenza tra la dichiarazione

```
int *p = &i;
```

e l'istruzione

```
p = &i;
```

Perché la dichiarazione p viene fatta precedere dal simbolo \*, mentre questo non succede nell'istruzione? [p. 256]

**R:** All'origine della confusione c'è il fatto che il simbolo \* può assumere diversi significati nel C, a seconda del contesto nel quale viene usato. Nella dichiarazione

```
int *p = &i;
```

il simbolo \* non rappresenta l'operatore indirection. Indica, invece, il tipo di p, informando il compilatore che p è un puntatore a un int. Quando compare in un'istruzione, invece, il simbolo \* esegue l'operazione di *indirection* (ovvero quando viene usato come operatore unario). L'istruzione

```
*p = &i; /*** SBAGLIATO ***/
```

sarebbe errata perché assegna l'indirizzo di i all'oggetto puntato da p e non allo stesso p.

**D:** C'è un modo per stampare l'indirizzo di una variabile? [p. 256]

**R:** Qualsiasi puntatore, incluso l'indirizzo di una variabile, può essere visualizzato chiamando la funzione printf e usando la specifica di conversione %p. Leggete la Sezione 22.3 per i dettagli.

**D:** La seguente dichiarazione è piuttosto confusa:

```
void f(const int *p);
```

**Indica forse che f non può modificare p? [p. 263]**

**R:** No. La dichiarazione specifica che f non possa modificare l'intero a cui p punta, mentre non impedisce a f di modificare la stessa variabile p.

```
void f(const int *p)
{
 int j;
 *p = 0; /** SBAGLIATO ***
 p = &j; /* ammesso */
}
```

Dato che gli argomenti vengono passati per valore, assegnarne uno nuovo alla variabile puntatore p (facendola puntare a qualcos'altro) non avrà alcun effetto al di fuori della funzione.

**D:** Quando dichiariamo un parametro di tipo puntatore, è possibile mettere la parola const di fronte al nome del parametro come succede nell'esempio seguente?

```
void f(int * const p);
```

**R:** Sì, sebbene l'effetto non sia lo stesso che avremmo avuto se la parola const avesse preceduto il tipo di p. Nella Sezione 11.4 abbiamo visto che mettere const *prima* del tipo di p protegge l'oggetto puntato da p. Mettere const *dopo* il tipo di p protegge lo stesso parametro p:

```
void f(int * const p)
{
 int j;
 p = 0; / ammissibile */
 p = &j; /** SBAGLIATO ***
}
```

Questa possibilità non viene sfruttata molto spesso. Dato che p è una semplice copia di un altro puntatore (l'argomento presente nell'invocazione della funzione), raramente vi sono ragioni per proteggerlo.

Ancora più rara è la necessità di proteggere sia p che l'oggetto a cui punta, cosa che può essere fatta mettendo const sia prima che dopo il tipo di p:

```
void f(const int * const p)
{
 int j;
 *p = 0; /** SBAGLIATO ***
 p = &j; /** SBAGLIATO ***
}
```

## Esercizi

**Sezione 11.2** 1. Se *i* è una variabile e *p* punta a *i*, quale delle seguenti espressioni sono degli alias per *i*?

- (a) \**p*      (c) \*&*p*      (e) \**i*      (g) \*&*i*
- (b) &*p*      (d) \*&*p*      (f) &*i*      (h) &\**i*

**Sezione 11.3** 2. Se *i* è una variabile int e *p* e *q* sono dei puntatori a int, quali dei seguenti assegnamenti sono validi?

- (a) *p* = 1;      (d) *p* = &*q*;      (g) *p* = \**q*;
- (b) *p* = &*i*;      (e) *p* = \*&*q*;      (h) \**p* = *q*;
- (c) &*p* = *q*;      (f) *p* = *q*;      (i) \**p* = \**q*;

**Sezione 11.4** 3. Ci si aspetta che la seguente funzione calcoli la somma e la media dei numeri contenuti nel vettore *a* di lunghezza *n*. I parametri *avg* e *sum* puntano alle variabili che devono essere modificate dalla funzione. Sfortunatamente la funzione contiene diversi errori. Trovateli e corregeteli.

```
void avg_sum(double a[], int n, double *avg, double *sum)
{
 int i;
 sum = 0.0;
 for (i = 0; i < n; i++)
 sum += a[i];
 avg = sum / n;
}
```

4. Scrivete la seguente funzione:

```
void swap(int *p, int *q);
```

La funzione *swap*, quando le vengono passati gli indirizzi di due variabili, deve scambiare i valori di queste ultime:

```
swap(&i, &j); /* scambia i valori di i e j */
```

5. Scrivete la funzione seguente:

```
void split_time(long total_sec, int *hr, int *min, int *sec);
```

*total\_sec* rappresenta un orario misurato come il numero di secondi dalla mezzanotte. I parametri *hr*, *min* e *sec* sono delle variabili puntatore nelle quali la funzione salverà l'orario equivalente espresso in ore (0 – 23), minuti (0 – 59) e secondi (0 – 59).

6. Scrivete la seguente funzione:

```
void find_two_largest(int a[], int n, int *largest, int *second_largest);
```

Quando le viene passato un vettore *a* di lunghezza *n*, la funzione deve cercare dentro a *a* il valore più grande e il secondo valore più grande. Questi devono essere salvati nelle variabili puntate rispettivamente da *largest* e *second\_largest*.

7. Scrivete la seguente funzione:

```
void split_date(int day_of_year, int year, int *month, int *day);
```

`day_of_year` è un intero compreso tra 1 e 366 che indica un particolare giorno dell'anno, `year` indica l'anno, mentre `month` e `day` puntano alle variabili nelle quali la funzione deve salvare rispettivamente il mese (1 – 12) e il giorno (1 – 31) equivalenti.

- Sezione 11.5** 8. Scrivete la seguente funzione:

```
int *find_largest(int a[], int n[]);
```

Quando viene passato un vettore `a` di lunghezza `n`, la funzione deve restituire un puntatore all'elemento più grande contenuto in `a`.

## Progetti di programmazione

1. Modificate il Progetto di programmazione 7 del Capitolo 2 in modo che includa la seguente funzione:

```
void pay_amount(int dollars, int *twenties, int *tens, int *fives, int *ones);
```

La funzione determina il minor numero di biglietti da 20 \$, 10 \$, 5 \$ e 1 \$ che sono necessari per pagare la somma rappresentata dal parametro `dollars`. Il parametro `twenties` punta a una variabile nella quale la funzione dovrà salvare il numero richiesto di biglietti da 20 \$. I parametri `tens`, `fives` e `ones` hanno funzioni analoghe.

2. Modificate il Progetto di programmazione 8 del Capitolo 5 in modo che includa la seguente funzione:

```
void find_closest_flight(int desired_time, int *departure_time, int *arrival_time);
```

Questa funzione dovrà trovare il volo il cui orario di partenza è il più vicino a quello contenuto in `desired_time` (espresso in minuti dalla mezzanotte). L'orario di partenza e quello di arrivo (anch'essi espressi in minuti dalla mezzanotte) dovranno essere salvati nelle variabili puntate rispettivamente da `departure_time` e `arrival_time`.

3. Modificate il Progetto di programmazione 3 del Capitolo 6 in modo che includa la seguente funzione:

```
void reduce(int numerator, int denominator, int *reduced_numerator, int *reduced_denominator);
```

I parametri `numerator` e `denominator` sono rispettivamente il numeratore e il denominatore di una frazione. I parametri `reduced_numerator` e `reduced_denominator` sono dei puntatori alle variabili nelle quali la funzione dovrà salvare il numeratore e il denominatore della frazione dopo che questa è stata ridotta ai minimi termini.

4. Modificate il programma `poker.c` della Sezione 10.5 spostando tutte le variabili esterne dentro il `main` e modificando le funzioni in modo che comunichino attraverso il passaggio degli argomenti. La funzione `analyze_hand` ha la necessità di modificare le variabili `straight`, `flush`, `four`, `three` e `pairs` e perciò le devono esser passati dei puntatori a queste ultime.

# 12 Puntatori e vettori

Il Capitolo 11 ha introdotto i puntatori e ha mostrato il loro utilizzo come argomenti per le funzioni e come valori restituiti dalle funzioni. Questo capitolo tratta un'altra applicazione dei puntatori. Il C permette di eseguire dell'aritmetica (addizioni e sottrazioni) sui puntatori che puntano a elementi di un vettore. Questo porta a un modo alternativo per elaborare i vettori nel quale i puntatori prendono il posto degli indici dei vettori stessi.

Come vedremo tra breve, in C vi è una stretta relazione tra puntatori e vettori. Sfrutteremo questa relazione nei prossimi capitoli, inclusi il Capitolo 13 (Stringhe) e il Capitolo 17 (Uso avanzato dei puntatori). Comprendere la connessione presente tra puntatori e vettori è fondamentale per padroneggiare pienamente il C: vi darà un'idea di come sia stato progettato il C e vi aiuterà a capire i programmi esistenti. Fate attenzione però al fatto che una delle ragioni principali per l'utilizzo dei puntatori nell'elaborazione dei vettori, ovvero l'efficienza, non è più così importante come in passato grazie all'evoluzione dei compilatori.

La Sezione 12.1 tratta l'aritmetica dei puntatori e mostra come essi possano essere confrontati utilizzando gli operatori relazionali e di uguaglianza. Successivamente la Sezione 12.2 dimostra come sia possibile usare l'aritmetica dei puntatori per elaborare gli elementi di un vettore. La Sezione 12.3 rivela una realtà chiave a riguardo dei vettori (il nome di un vettore può fare le veci di un puntatore al primo elemento) e illustra come funzionano veramente i parametri costituiti da vettori. La Sezione 12.4 illustra come gli argomenti delle prime tre sezioni si applichino ai vettori multidimensionali. La Sezione 12.5 chiude il capitolo esaminando la relazione presente tra i puntatori e i vettori a lunghezza variabile caratteristici del C99.

## 12.1 Aritmetica dei puntatori

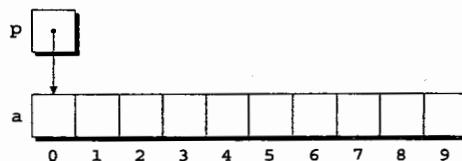
Nella Sezione 11.5 abbiamo visto che i puntatori possono puntare agli elementi di un vettore. Per esempio: supponete che `a` e `p` siano stati dichiarati nel modo seguente:

```
int a[10], *p;
```

possiamo fare in modo che `p` punti ad `a[0]` scrivendo

```
p = &a[0];
```

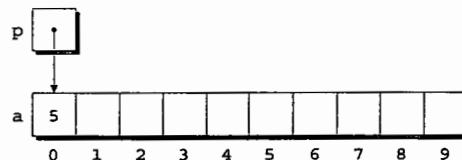
Graficamente ecco quello che abbiamo fatto:



Adesso possiamo accedere ad *a[0]* attraverso *p*. Per esempio possiamo memorizzare il valore 5 all'interno di *a[0]* scrivendo

```
*p = 5;
```

Ecco come si presenta ora la nostra figura:



Far sì che un puntatore *p* punti a un elemento del vettore *a* non è poi così interessante. Tuttavia, effettuando operazioni di **aritmetica dei puntatori** (o **aritmetica degli indirizzi**) su *p*, possiamo accedere agli altri elementi di *a*. Il C supporta tre (o solo tre) forme di aritmetica dei puntatori:

Sommare un intero a un puntatore.

Sottrarre da un puntatore un intero.

Sottrarre da un puntatore un altro puntatore.

Vediamo ognuna di queste operazioni. I nostri esempi assumono la presenza delle seguenti dichiarazioni:

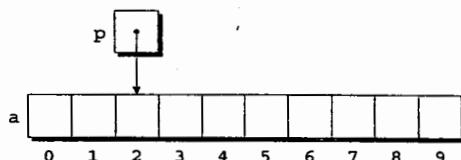
```
int a[10], *p, *q, i;
```

## Sommare un intero a un puntatore

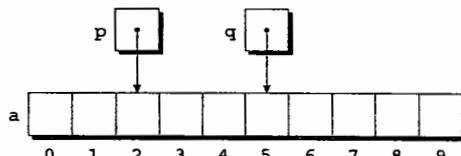
Sommare un intero *j* a un puntatore *p* restituisce un puntatore all'elemento che si trova *j* posizioni dopo dell'elemento puntato originariamente da *p*. Più precisamente se *p* punta all'elemento *a[i]* allora *p + j* punta all'elemento *a[i + j]* (ammesso, ovviamente, che *a[i+j]* esista).

L'esempio seguente illustra la somma ai puntatori, le figure mostreranno i valori assunti da *p* e *q* in vari momenti durante l'esecuzione.

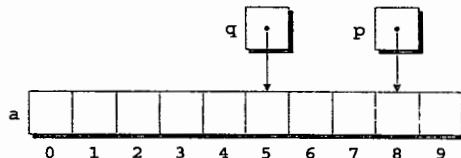
`p = &a[2];`



`q = p + 3;`



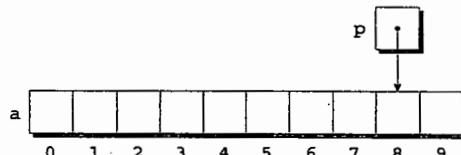
`p += 6;`



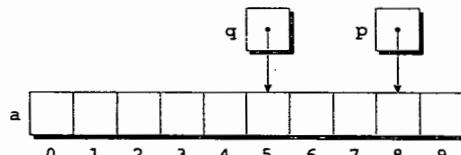
## Sottrarre un intero da un puntatore

Se `p` punta all'elemento `a[i]` di un vettore, allora `p - j` punta ad `a[i - j]`. Per esempio:

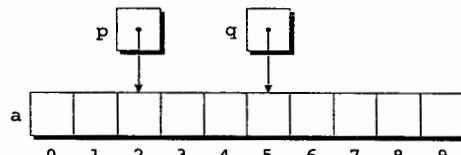
`p = &a[8];`



`q = p - 3;`



`p -= 6;`

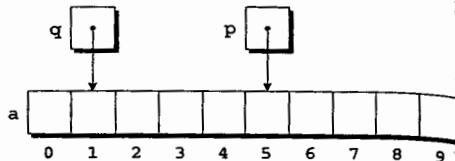


## Sottrarre da un puntatore un altro puntatore

Quando si sottrae un puntatore da un altro, il risultato consiste nella distanza tra i due puntatori (misurata in elementi del vettore). Quindi se `p` punta ad `a[i]` e `q` punta ad `a[j]`, allora `p - q` è uguale a  $i - j$ . Per esempio:

```
p = &a[5];
q = &a[1];
```

```
i = p - q; /* i è uguale a 4 */
i = q - p; /* i è uguale a -4 */
```



**!** Eseguire calcoli su un puntatore che non punta a un elemento di un vettore provoca un comportamento indefinito. Inoltre, anche l'effetto della sottrazione tra due puntatori non è definito se questi non puntano a elementi dello stesso vettore.

## Confrontare i puntatori

Possiamo confrontare i puntatori utilizzando gli operatori relazionali (`<`, `<=`, `>`, `>=`) e gli operatori di uguaglianza (`==` e `!=`). Naturalmente usare gli operatori relazionali per confrontare due puntatori ha senso solamente nel caso in cui entrambi i puntatori puntino a elementi dello stesso vettore. Il risultato del confronto dipende dalla posizione relativa dei due elementi all'interno del vettore. Per esempio, dopo gli assegnamenti

```
p = &a[5];
q = &a[1];
```

il valore di `p <= q` è 0 e il valore di `p >= q` è 1.

## C99 Puntatori a letterali composti

Per un puntatore è anche possibile puntare a un elemento presente all'interno di un vettore creato con un letterale composto [letterale composto > 9.3]. Ricordate che i letterali composti sono una funzionalità del C99 che può essere usata per creare un vettore privo di nome.

Considerate l'esempio seguente:

```
int *p = (int []){3, 0, 3, 4, 1};
```

`p` punta al primo dei cinque elementi di un vettore contenente gli interi 3, 0, 3, 4 e 1. Utilizzare un letterale composto ci risparmia la fatica di dover dichiarare una variabile vettore e far sì che `p` punti al primo elemento di questa:

```
int a[] = {3, 0, 3, 4, 1};
int *p = &a[0];
```

## 12.2 Usare i puntatori per l'elaborazione dei vettori

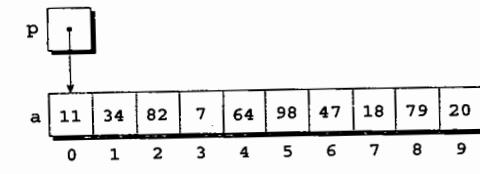
L'aritmetica dei puntatori ci permette di visitare tutti gli elementi di un vettore incrementando ripetutamente una variabile puntatore. Il seguente frammento di programma, che somma gli elementi del vettore `a`, illustra questa tecnica. In questo esempio la variabile `p` punta inizialmente ad `a[0]`. A ogni iterazione del ciclo la variabile `p` viene

incrementata, si ha così che questa punti ad  $a[1]$ , poi ad  $a[2]$  e così via. Il ciclo termina quando  $p$  oltrepassa l'ultimo elemento di  $a$ .

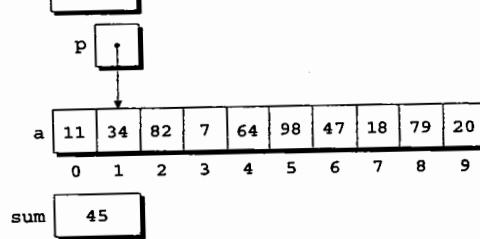
```
#define N 10
-
int a[N], sum, *p;
-
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
 sum += *p;
```

Le immagini riportate di seguito illustrano il contenuto delle variabili  $a$ ,  $sum$  e  $p$  alla fine delle prime tre iterazioni (prima che  $p$  venga incrementato).

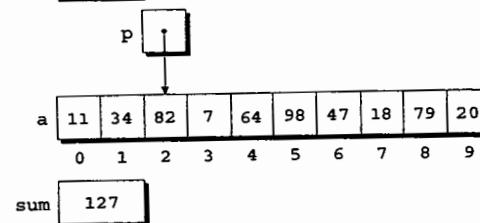
Alla fine della prima iterazione



Alla fine della seconda iterazione



Alla fine della terza iterazione



La condizione  $p < &a[N]$  presente nel ciclo for merita un cenno. Sebbene possa sembrare strano è possibile applicare l'operatore  $&$  ad  $a[N]$  anche se questo elemento non esiste (il vettore  $a$  ha indici che vanno da 0 a  $N - 1$ ). Utilizzare in questo modo  $a[N]$  è perfettamente sicuro visto che il ciclo non cerca di esaminare il suo valore. Il corpo del ciclo viene eseguito per  $p$  uguale a  $&a[0]$ ,  $&a[1]$ , ...,  $&a[N - 1]$ , ma quando  $p$  diventa uguale a  $&a[N]$  il ciclo si ferma.

Avremmo potuto scrivere facilmente lo stesso ciclo senza i puntatori, utilizzando al loro posto gli indici. L'argomento più gettonato in supporto dell'aritmetica dei puntatori dipende dal fatto che questi possono risparmiare tempo di esecuzione. Tuttavia questo dipende dall'implementazione (attualmente alcuni compilatori C producono codice migliore per i cicli for che si affidano all'indicizzazione).

## Abbinare gli operatori \* e ++

Spesso i programmati C abbinano l'uso degli operatori `*` (*indirection*) e `++` all'interno delle istruzioni che elaborano gli elementi dei vettori. Considerate il semplice caso del salvataggio di un valore all'interno di un elemento di un vettore seguito dall'avanzamento all'elemento successivo. Utilizzando l'indicizzazione potremmo scrivere

```
a[1++] = j;
```

se `p` è un puntatore a un elemento del vettore, l'istruzione corrispondente sarebbe

```
*p++ = j;
```

A causa della precedenza della versione a suffisso di `++` rispetto all'operatore `*`, il compilatore interpreta l'istruzione come

```
*(p++) = j;
```

Il valore di `p++` è `p` (visto che stiamo usando la versione a suffisso di `++`, `p` non viene incrementato fino a quando l'espressione non viene calcolata). Di conseguenza il valore di `*(p++)` è uguale a `*p`, l'oggetto al quale sta puntando `p`.

Naturalmente `*p++` non è l'unica combinazione degli operatori `*` e `++`. Per esempio possiamo scrivere `(*p)++` che restituisce il valore dell'oggetto puntato da `p` e successivamente incrementa l'oggetto in questione (`p` non viene modificata). La tabella seguente chiarisce quanto detto.

| <i>Espresione</i>               | <i>Significato</i>                                                                                                      |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| <code>*p++ oppure *(p++)</code> | Prima dell'incremento il valore dell'espressione è <code>*p</code> , successivamente viene incrementata <code>p</code>  |
| <code>(*p)++</code>             | Prima dell'incremento il valore dell'espressione è <code>*p</code> , successivamente viene incrementato <code>*p</code> |
| <code>++*p oppure *(++p)</code> | Prima viene incrementata <code>p</code> , dopo l'incremento il valore dell'espressione è <code>*p</code>                |
| <code>++*p oppure ++(*p)</code> | Prima viene incrementato <code>*p</code> , dopo l'incremento il valore dell'espressione è <code>*p</code>               |

Nei programmi potrete trovare tutte e quattro le espressioni, sebbene alcune siano molto più frequenti di altre. Quella che vedremo più di frequente è `*p++`, un'espressione molto comoda nei cicli. Per sommare tutti gli elementi, invece di scrivere

```
for (p = &a[0]; p < &a[N]; p++)
 sum += *p;
```

potremmo scrivere

```
p = &a[0];
while (p < &a[N])
 sum += *p++;
```

Gli operatori `*` e `--` si combinano allo stesso modo visto per `*` e `++`. Per un'applicazione che combini `*` e `--` ritorniamo all'esempio della Sezione 10.2. La versione originale dello stack si basa su una variabile intera chiamata `top` che tiene traccia della posizione della cima dello stack nel vettore `contents`. Rimpiazziamo `top` con una variabile puntatore che punta inizialmente all'elemento 0 del vettore:

```
int *top_ptr = &contents[0];
```

Ecco le due nuove funzioni push e pop (l'aggiornamento delle altre funzioni dello stack viene lasciato come esercizio):

```
void push(int i)
{
 if (is_full())
 stack_overflow();
 else
 *top_ptr++ = i;
}
int pop(void)
{
 if (is_empty())
 stack_underflow();
 else
 return *--top_ptr;
}
```

Osservate che è scritto `--top_ptr` e non `*top_ptr--` dato che si vuole che pop decrementi `top_ptr` prima di caricare il valore al quale punta.

## 12.3 Usare il nome di un vettore come puntatore

L'aritmetica dei puntatori non è l'unico collegamento esistente tra i vettori e i puntatori. Ecco un'altra relazione chiave: *il nome di un vettore può essere usato come un puntatore al primo elemento del vettore*. Questa relazione semplifica l'aritmetica dei puntatori e rende più versatili sia i vettori che i puntatori.

Per esempio, supponete che il vettore `a` venga dichiarato come segue:

```
int a[10];
```

Possiamo modificare `a[0]` usando `a` come un puntatore al primo elemento del vettore:

```
a = 7; / salva 7 in a[0] */
```

Possiamo modificare `a[1]` attraverso il puntatore `a + 1`:

```
(a+1) = 12; / salva 12 in a[1] */
```

In generale, la scrittura `a + i` è equivalente a `&a[i]` (entrambe rappresentano un puntatore all'elemento `i`-esimo di `a`) mentre `*(a+i)` è equivalente a `a[i]` (entrambe rappresentano l'elemento `i`-esimo). In altre parole, l'indicizzazione di un vettore può essere vista come una forma di aritmetica dei puntatori.

Il fatto che il nome di un vettore possa essere usato come un puntatore facilita la scrittura dei cicli che visitano un vettore. Considerate il seguente ciclo preso dalla Sezione 12.2:

```
for (p = &a[0]; p < &a[N]; p++)
 sum += *p;
```

Per semplificare il ciclo possiamo sostituire `&a[0]` con `a` e `&a[N]` con `a + N`:

```
for (p = a; p < a + N; p++)
 sum += *p;
```



Sebbene il nome di un vettore possa essere utilizzato come un puntatore, non è possibile assegnargli un nuovo valore. Cercare di farlo puntare altrove è un errore:

```
while (*a != 0)
 a++; /* SBAGLIATO */
```

Non è un problema di cui preoccuparsi, possiamo sempre copiare `a` in una variabile puntatore e poi modificare quest'ultima:

```
p = a;
while (*p != 0)
 p++;
```

#### PROGRAMMA

## Invertire una sequenza di numeri (rivisitato)

Il programma `reverse.c` della Sezione 8.1 legge 10 numeri e poi li scrive in ordine inverso. Quando il programma legge i numeri li salva in un vettore. Una volta che tutti i numeri sono stati letti, il programma, per stampare i numeri, ripercorre in senso inverso il vettore.

Il programma originale utilizza l'indicizzazione per accedere agli elementi del vettore. Ecco una nuova versione nella quale l'indicizzazione viene sostituita dall'aritmetica dei puntatori.

```
reverse3.c /* Inverte una sequenza di numeri (versione con i puntatori) */
#include <stdio.h>

#define N 10

int main(void)
{
 int a[N], *p;

 printf("Enter %d numbers: ", N);
 for (p = a; p < a + N; p++)
 scanf("%d", p);

 printf("In reverse order:");
 for (p = a + N - 1; p >= a; p--)
 printf(" %d", *p);
 printf("\n");

 return 0;
}
```

Nel programma originale la variabile intera *i* tiene traccia della posizione corrente all'interno del vettore. La nuova versione sostituisce *i* con *p*, una variabile puntatore. I numeri sono ancora memorizzati in un vettore, stiamo semplicemente usando una tecnica diversa per tenere traccia del punto interno al vettore nel quale ci troviamo.

Osservate che il secondo argomento della *scanf* è *p* e non *&p*. Dato che *p* punta all'elemento di un vettore, questo lo rende un argomento soddisfacente per la *scanf*, al contrario *&p* sarebbe un puntatore a un puntatore di un elemento del vettore.

## Argomenti costituiti da vettori (rivisitato)

Quando viene passato a una funzione, il nome di un vettore viene sempre trattato come un puntatore. Considerate la funzione seguente che restituisce il più grande tra gli elementi presenti in un vettore di interi:

```
int find_largest(int a[], int n)
{
 int i, max;
 max = a[0];
 for (i = 1; i < n; i++)
 if(a[i] > max)
 max = a[i];
 return max;
}
```

Supponete di invocare la funzione *find\_largest* in questo modo:

```
largest = find_largest(b, N);
```

Questa chiamata fa sì che ad *a* venga assegnato un puntatore al primo elemento di *b*: il vettore di per sé non viene copiato.

Il fatto che un argomento costituito da un vettore venga trattato come un puntatore ha importanti conseguenze.

- Quando a una funzione viene passata una variabile ordinaria il suo valore viene copiato e nessuna modifica al parametro corrispondente ha effetti su di essa. Al contrario un vettore utilizzato come argomento non è protetto da modifiche. Dato che non viene effettuata una sua copia. Per esempio: la funzione seguente (che abbiamo visto per la prima volta nella Sezione 9.3) modifica il vettore ponendo a zero tutti i suoi elementi:

```
void store_zeros(int a[], int n)
{
 int i;
 for (i = 0; i < n; i++)
 a[i] = 0;
}
```

Per indicare che un vettore non dovrà essere modificato possiamo includere la parola *const* all'interno della dichiarazione:

```
int find_largest(const int a[], int n)
{
 ...
}
```

Se `const` è presente, il compilatore controllerà che nel corpo della funzione `find_largest` non venga fatto nessun assegnamento a elementi di `a`.

- Il tempo richiesto per passare un vettore a una funzione non dipende dalla dimensione del vettore. Visto che non ne viene effettuata la copia, non ci sono svantaggi nel passare vettori di grandi dimensioni.
- Un parametro costituito da un vettore può essere dichiarato come puntatore se lo si volesse. La funzione `find_largest`, per esempio, poteva essere definita in questo modo:

```
int find_largest(int *a, int n)
{
 ...
}
```

Aver dichiarato che `a` è un puntatore è equivalente ad averlo dichiarato come un vettore. Il compilatore gestisce le due dichiarazioni come se fossero identiche.



Sebbene dichiarare un *parametro* come vettore o come puntatore sia la stessa cosa, questo non è assolutamente vero per una *variabile*. La dichiarazione

```
int a[10];
```

fa sì che il compilatore riservi dello spazio per 10 interi. Al contrario la dichiarazione

```
int *a;
```

fa sì che il compilatore allochi dello spazio per una variabile puntatore. Nell'ultimo caso `a` non è un vettore e quindi cercare di utilizzarlo in quel modo porterebbe a delle conseguenze disastrose. Per esempio, l'assegnamento

```
*a = 0; /*** SBAGLIATO ***/
```

andrebbe a memorizzare uno zero nella locazione puntata da `a`. Dato che non sappiamo dove `a` stia puntando, il programma avrà un comportamento indefinito.

- A una funzione avente un parametro dichiarato come vettore può essere passata una "fetta" di un vettore (una sequenza di elementi consecutivi). Supponete di volere che `find_largest` trovi il più grande elemento presente in una porzione del vettore `b`, diciamo `b[5], ..., b[14]`. Al momento dell'invocazione di `find_largest` le passeremo l'indirizzo di `b[5]` e il numero 10 indicando così la nostra volontà che la funzione esamini 10 elementi del vettore a partire da `b[5]`:

```
largest = find_largest(&b[5], 10);
```

## Utilizzare un puntatore come il nome di un vettore

Dato che il C permette di utilizzare il nome di un vettore come se fosse un puntatore, possiamo anche indicizzare un puntatore come se fosse il nome di un vettore? Ormai ci possiamo aspettare che la risposta sia positiva e in effetti è così. Ecco un esempio:

```
#define N 10

int a[N], i, sum = 0, *p = a;

for (i = 0; i < N; i++)
 sum += p[i];
```

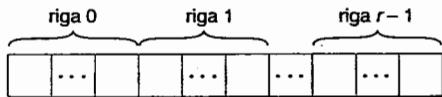
Il compilatore gestisce `p[i]` come se fosse `*(p+i)` che è un modo assolutamente lecito di utilizzare l'aritmetica dei puntatori. Sebbene la possibilità di indicizzare un puntatore sembri poco più di una curiosità, vedremo nella Sezione 17.3 che è piuttosto utile.

## 12.4 Puntatori e vettori multidimensionali

I puntatori, così come possono puntare agli elementi di un vettore a una dimensione, possono anche puntare agli elementi di un vettore multidimensionale. In questa sezione esamineremo delle comuni tecniche di utilizzo dei puntatori per l'elaborazione dei vettori multidimensionali. Per semplicità ci atterremo ai vettori bidimensionali, ma tutto quello che faremo si applica allo stesso modo ai vettori con un numero maggiore di dimensioni.

### Elaborare gli elementi di un vettore multidimensionale

Abbiamo visto nella Sezione 8.2 che il C memorizza i vettori bidimensionali ordinandoli per riga. In altre parole prima vengono inseriti gli elementi della riga 0, poi quelli della riga 1 e così via. Un vettore di  $r$  righe si presenta in questo modo:



Possiamo sfruttare questa disposizione lavorando con i puntatori. Se facciamo in modo che il puntatore `p` punti al primo elemento del vettore bidimensionale (l'elemento presente alla riga 0 e alla colonna 0), allora incrementando `p` ripetutamente possiamo visitare tutti gli elementi del vettore stesso.

Come esempio guardiamo al problema dell'inizializzazione a 0 di tutti gli elementi di un vettore bidimensionale. Supponete che il vettore venga dichiarato in questo modo:

```
int a[NUM_ROWS][NUM_COLS];
```

La tecnica più ovvia sarebbe quella di utilizzare dei cicli `for` annidati:

```
int row, col;
for (row = 0; row < NUM_ROWS; row++)
 for (col = 0; col < NUM_COLS; col++)
 a[row][col] = 0;
```

Tuttavia se vediamo a come un vettore unidimensionale di interi (che è il modo in cui è memorizzato), possiamo rimpiazzare la coppia di cicli con un ciclo solo:

```
int *p;
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
 *p = 0;
```

Il ciclo inizia con `p` che punta ad `a[0][0]`. I successivi incrementi di `p` lo fanno puntare ad `a[0][1], a[0][2], a[0][3]` e così via. Quando `p` raggiunge `a[0][NUM_COLS-1]` (l'ultimo elemento della riga 0) il successivo incremento lo fa puntare ad `a[1][0]`, il primo elemento della riga 1. Il processo continua fino a che `p` va oltre ad `a[NUM_ROWS-1][NUM_COLS-1]`, l'ultimo elemento del vettore.

Sebbene trattare i vettori a due dimensioni come se fossero dei normali vettori unidimensionali sembri un piccolo trucco, questa tecnica funziona con la maggior parte dei compilatori C. Se poi questa sia una buona pratica o meno è un'altra questione. Tecniche come quella appena presentata si scontrano con la leggibilità del programma ma in compenso (almeno con alcuni vecchi compilatori), portano a un incremento dell'efficienza. Tuttavia, per molti compilatori moderni il vantaggio in termini di velocità del codice sono minimi o inesistenti.

D&R

## Elaborare le righe di un vettore multidimensionale

Cosa succede se vogliamo elaborare gli elementi presenti in *una sola* riga di un vettore bidimensionale? Anche questa volta abbiamo la possibilità di utilizzare la variabile puntatore `p`. Per visitare gli elementi della riga `i` dovremo inizializzare `p` in modo che punti all'elemento 0 di quella riga:

```
p = &a[i][0];
```

o più semplicemente possiamo scrivere

```
p = a[i];
```

dato che per un qualsiasi vettore bidimensionale `a`, l'espressione `a[i]` è un puntatore al primo elemento della riga `i`. Per capire perché ciò funzioni ricordatevi la "formula magica" che lega l'indicizzazione dei vettori all'aritmetica dei puntatori: per un vettore `a`, l'espressione `a[i]` è equivalente a `*(a + i)`. Di conseguenza `&a[i][0]` è lo stesso che scrivere `&(*(a[i] + 0))`, che è equivalente a `8*a[i]`, che a sua volta lo è ad `a[i]` in quanto gli operatori `&` e `*` si annullano a vicenda. Utilizzeremo questa semplificazione nel ciclo seguente che impone a zero gli elementi del vettore `a`:

```

int a[NUM_ROWS][NUM_COLS], *p, i;

for (p = a[i]; p < a[i] + NUM_COLS; p++)
 *p = 0;

```

Considerato che `a[i]` è un puntatore alla riga `i` del vettore `a`, possiamo anche passarlo a funzioni che si aspettano un vettore unidimensionale come argomento. In altre parole, una funzione che sia stata progettata per lavorare con un vettore unidimensionale, può falso anche con una riga appartenente a un vettore bidimensionale. Come risultato si ha che funzioni come `find_largest` e `store_zeros` sono più versatili di quello che potreste aspettarvi. Tenete presente che, in origine, la funzione `find_largest` era stata sviluppata per trovare l'elemento più grande presente in un vettore, tuttavia possiamo facilmente utilizzarla per trovare l'elemento più grande tra quelli della riga `i` del vettore bidimensionale `a`:

```
largest = find_largest(a[i], NUM_COLS);
```

## Elaborare le colonne di un vettore multidimensionale

Elaborare gli elementi di una *colonna* di un vettore bidimensionale non è facile a causa del fatto che questi vengono memorizzati per righe e non per colonne. Ecco un ciclo che azzerà la colonna `i` del vettore `a`:

```

int a[NUM_ROWS][NUM_COLS], (*p)[NUM_COLS], i;

for (p = &a[0]; p < &a[NUM_ROWS]; p++)
 (*p)[i] = 0;

```

`p` è stato dichiarato come puntatore a un vettore di interi di lunghezza `NUM_COLS`. Le parentesi attorno a `*p` in `(*p)[NUM_COLS]` sono necessarie. Senza di esse il compilatore tratterebbe `p` come un vettore di puntatori invece che un puntatore a un vettore. L'espressione `p++` fa avanzare `p` all'inizio della riga successiva. Nell'espressione `(*p)[i]`, `*p` rappresenta un intera riga di `a` e quindi `(*p)[i]` seleziona l'elemento della colonna `i` di quella riga. La parentesi in `(*p)[i]` sono essenziali perché altrimenti il compilatore interpreterebbe `*p[i]` come `*(p[i])`.

## Utilizzare il nome di un vettore multidimensionale come puntatore

Proprio come per i vettori unidimensionali è possibile utilizzare il nome del vettore stesso come un puntatore, questo succede per tutti i vettori indipendentemente dalla loro dimensione. Nonostante ciò è necessaria una certa attenzione nel farlo. Considerate il seguente vettore:

```
int a[NUM_ROWS][NUM_COLS];
```

a non è un puntatore ad `a[0][0]`, ma un puntatore ad `a[0]`. Questo ha più senso se lo guardiamo dal punto di vista del C, il quale considera `a` come un vettore bidimensionale, bensì come un vettore unidimensionale i cui elementi sono a loro volta dei vettori unidimensionali. Quando viene usato come un puntatore, `a` è del tipo `int`

`(*)[NUM_COLS]` (puntatore a un vettore di interi di lunghezza `NUM_COLS`). Sapere che `a` punta ad `a[0]` è utile per semplificare i cicli che elaborano gli elementi di un vettore bidimensionale. Per esempio, per azzerare la colonna `i` del vettore `a`, invece di scrivere

```
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
 (*p)[i] = 0;
```

possiamo scrivere

```
for (p = a; p < a + NUM_ROWS; p++)
 (*p)[i] = 0;
```

Un'altra situazione nella quale questa nozione torna utile si presenta quando vogliamo "ingannare" una funzione per farle credere che un vettore multidimensionale sia in realtà unidimensionale. Per esempio: considerate come potremmo utilizzare `find_largest` per cercare l'elemento più grande di `a`. Proviamo a passare a (`l'indirizzo del vettore`) come primo argomento di `find_largest`, mentre come secondo argomento passeremo `NUM_ROWS * NUM_COLS` (il numero totale degli elementi di `a`):

```
largest = find_largest(a, NUM_ROWS * NUM_COLS); /** SBAGLIATO **/
```

Sfortunatamente il compilatore non accetterà questa istruzione perché il tipo di `a` è `int (*)[NUM_COLS]` mentre `find_largest` si aspetta un argomento del tipo `int *`. La chiamata corretta è:

```
largest = find_largest(a[0], NUM_ROWS * NUM_COLS);
```

**D&R** `a[0]` punta all'elemento 0 della riga 0 ed è del tipo `int *` (dopo la conversione effettuata dal compilatore) e quindi la seconda invocazione funzionerà correttamente.

## C99 12.5 Puntatori e vettori a lunghezza variabile

Ai puntatori è permesso puntare agli elementi dei vettori a lunghezza variabile (VLA) [[vettori a lunghezza variabile > 8.3](#)]. Un normale puntatore può essere usato anche per puntare a un elemento di un VLA unidimensionale:

```
void f(int n)
{
 int a[n], *p;
 p = a;
 ...
}
```

Quando un VLA ha più di una dimensione, il tipo del puntatore dipende dalla lunghezza di ogni dimensione a eccezione della prima. Analizziamo il caso bidimensionale:

```
void f(int m, int n)
{
 int a[m][n], (*p)[n];
 p = a;
 ...
}
```

Dato che il tipo *p* dipende da *n*, la quale non è costante, si dice che *p* sia di un **tipo modificato dinamicamente**. Osservate che la validità di un assegnamento come *p* = *a* non può essere sempre determinato dal compilatore. Per esempio, il codice seguente sarebbe compilabile sebbene sia corretto solo nel caso in cui *m* ed *n* sono uguali:

```
int a[m][n], (*p)[m];
p = a;
```

se *m* è diverso da *n* qualsiasi successivo utilizzo di *p* causerebbe un comportamento indefinito.

I tipi modificati dinamicamente sono soggetti ad alcune restrizioni esattamente così come lo sono i vettori a lunghezza variabile. La restrizione più importante è che le dichiarazioni di tipi modificati dinamicamente devono risiedere nel corpo di una funzione o nel prototipo di una funzione.

L'aritmetica dei puntatori funziona per iVLA esattamente come per i vettori normali. Ritorniamo all'esempio della Sezione 12.4 che si occupa di azzerare una singola colonna di un vettore bidimensionale *a*, ma questa volta dichiariamo quest'ultimo come unVLA:

```
int a[m][n];
```

Un puntatore in grado di puntare a una riga dovrebbe essere dichiarato in questo modo:

```
int (*p)[n];
```

Il ciclo che azzerà la colonna *i* è quasi identico a quello utilizzato nella Sezione 12.4:

```
for (p = a; p < a + m; p++)
 (*p)[i] = 0;
```

## Domande & Risposte

**D: Non capiamo l'aritmetica dei puntatori. Se un puntatore è un indirizzo, questo significa che un'espressione come *p* + *j* somma *j* all'indirizzo contenuto in *p*? [p. 270]**

**R:** No. Gli interi usati nell'aritmetica dei puntatori vengono scalati a seconda del tipo del puntatore. Se per esempio *p* è di tipo int \*, allora *p* + *j* tipicamente somma a *p* il valore  $4 \times j$  (assumendo che gli int vengano rappresentati con 4 byte). Se invece *p* è di tipo double \*, allora *p* + *j* probabilmente sommerà a *p* il valore  $8 \times j$ , dato che i valori double di solito sono lunghi 8 byte.

**D: Quando si scrive un ciclo per elaborare un vettore, è meglio utilizzare l'indicizzazione del vettore o l'aritmetica dei puntatori? [p. 273]**

**R:** Questa domanda non ha una risposta semplice visto che dipende dalla macchina che state usando e dallo stesso compilatore. Agli albori del C sul PDP-11, l'aritmetica dei puntatori conduceva a programmi più veloci. Sulle macchine odierne, con i moderni compilatori, spesso l'indicizzazione è una tecnica altrettanto buona, se non migliore. È opportuno imparare entrambe le tecniche e poi usare quella che sembra più naturale per il tipo di programma che si sta scrivendo.

**\*D: Da qualche parte abbiamo letto che scrivere  $i[a]$  equivale a scrivere  $a[i]$ . Questo è vero?**

**R:** Sì, lo è, sebbene sia piuttosto strano. Il compilatore tratta  $i[a]$  come  $*(i + a)$  che è equivalente  $*(a + i)$  (la somma dei puntatori, come quella ordinaria, è commutativa). Ma  $*(a + i)$  è a sua volta equivalente a  $a[i]$ , il che era quanto si voleva dimostrare. Tuttavia, è preferibile non utilizzare  $i[a]$  all'interno dei programmi a meno che non si stia pianificando di partecipare alla prossima competizione di "Obfuscated C".

**D: Perché nella dichiarazione di un parametro  $*a$  è equivalente ad  $a[]$ ? [p. 278]**

**R:** Entrambi indicano che ci si aspetta che l'argomento sia un puntatore. Le medesime operazioni su  $a$  sono possibili in entrambi i casi (in particolare l'aritmetica dei puntatori e l'indicizzazione dei vettori). Inoltre in entrambi i casi all'interno della funzione è possibile assegnare un nuovo valore ad  $a$  (sebbene il C ci permetta di utilizzare il nome di una *variabile* vettore solo come un "puntatore costante", non c'è questa restrizione sul nome di un *parametro* costituito da un vettore).

**D: È uno stile migliore dichiarare un vettore come  $*a$  o come  $a[]$ ?**

**R:** Questa è una domanda difficile. Da un punto certo di vista,  $a[]$  è la scelta ovvia visto che  $*a$  è ambiguo (la funzione vuole un vettore di oggetti o un puntatore a un singolo oggetto?). D'altro canto molti programmati sostengono che dichiarare il parametro come  $*a$  è più accurato visto che ci ricorda che viene passato solamente un puntatore e non una copia del vettore. Altri programmati impiegano  $*a$  o  $a[]$  a seconda che la funzione faccia uso dell'indicizzazione del vettore o dell'aritmetica dei puntatori per accedere agli elementi del vettore (questo è l'approccio che verrà usato dal libro). Nella pratica  $*a$  è più comune di  $a[]$  quindi sarebbe meglio che vi abituaste a usarlo. Per quel che può significare, Dennis Ritchie attualmente si riferisce alla notazione  $a[]$  come a un "fossile vivente" che "serve sia per confondere il principiante che per allarmare il lettore".

**D: Abbiamo visto che nel C i vettori e i puntatori sono strettamente legati. Sarebbe accurato dire che sono intercambiabili?**

**R:** No. È vero che i *parametri* vettore sono intercambiabili con i parametri puntatore, tuttavia le *variabili* non sono equivalenti alle variabili puntatore. Tecnicamente il nome di un vettore non è un puntatore, il compilatore C lo *converte* in un puntatore quando è necessario. Per capire meglio questa differenza, considerate quello che succede quando applichiamo l'operatore *sizeof* al vettore  $a$ . Il valore di *sizeof(a)* è pari al numero totale di byte presenti nel vettore, la dimensione di ogni elemento moltiplicato per il numero di elementi. Tuttavia se  $p$  è una variabile puntatore, *sizeof(p)* è il numero di byte richiesto per salvare un valore puntatore.

**D: Lei ha detto che trattare un vettore bidimensionale come un vettore a una dimensione funziona con la maggior parte dei compilatori C. Non funziona con tutti i compilatori? [p. 280]**

**R:** No. Alcuni moderni compilatori "bound-checking" tengono traccia non solo del tipo di un puntatore ma, quando questo punta a un vettore, anche della lunghezza di quest'ultimo. Per esempio, supponete che a  $p$  venga assegnato un puntatore ad  $a[0][0]$ . Tecnicamente  $p$  punta al primo elemento di  $a[0]$ , ovvero un vettore unidimensionale. Se incrementiamo ripetutamente  $p$  in modo da visitare tutti gli elementi di  $a$ , andremo al

di fuori dei limiti una volta che *p* oltrepassa l'ultimo elemento di *a[0]*. Un compilatore che esegue il controllo dei limiti può inserire del codice per controllare che *p* venga usato solo per accedere agli elementi presenti nel vettore puntato da *a[0]*. Un tentativo di incrementare *p* oltre la fine di questo vettore verrebbe considerato come un errore.

**D:** Se *a* è un vettore bidimensionale, perché *a* `find_largest` passiamo *a[0]* invece dello stesso *a*? Non puntano entrambi alla stessa locazione ovvero l'inizio del vettore? [p. 282]

**R:** In effetti entrambi puntano all'elemento *a[0][0]*. Il problema è che *a* è del tipo sbagliato, infatti quando viene usato come argomento è un puntatore a un vettore. La funzione `find_largest` invece si aspetta un puntatore a un intero. Tuttavia *a[0]* è di tipo `int *` e quindi non è un argomento accettabile per la funzione. Tutta questa preoccupazione riguardo ai tipi in effetti è un bene, se il C non fosse così pignolo potremmo commettere ogni sorta di errori con i puntatori senza che il compilatore se ne accorga.

## Esercizi

### Sezione 12.1

- Supponete che siano state effettuate le seguenti dichiarazioni:

```
int a[] = {5, 15, 34, 54, 14, 2, 52, 72};
int *p = &a[1], *q = &a[5];
```

- (a) Qual è il valore di `*(p+3)`?
- (b) Qual è il valore di `*(q-3)`?
- (c) Qual è il valore di `q-p`?
- (d) La condizione `p < q` è vera o falsa?
- (e) La condizione `*p < *q` è vera o falsa?

- W 2. Supponete che `high`, `low` e `middle` siano tutte variabili puntatori dello stesso tipo e che `low` e `high` puntino a elementi di un vettore. Perché l'istruzione seguente non è lecita e come può essere corretta?

```
middle = (low + high) / 2;
```

### Sezione 12.2

- Quali saranno gli elementi del vettore *a* dopo che le seguenti istruzioni sono state eseguite?

```
#define N 10
```

```
int a[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *p = &a[0], *q = &a[N-1], temp;

while (p < q) {
 temp = *p;
 *p++ = *q;
 *q-- = temp;
}
```

- W 4. Riscrivete le funzioni `make_empty`, `is_empty` e `is_full` della Sezione 10.2 in modo da usare la variabile puntatore `top_ptr` al posto della variabile intera `top`.

### Sezione 12.3

- Supponete che *a* sia un vettore unidimensionale e che *p* sia una variabile puntatore. Assumete che sia appena stato eseguito l'assegnamento *p* = *a*. Quale delle se-

uenti istruzioni è illecita a causa dei tipi non adatti? Delle espressioni rimanenti quali sono vere (hanno valore diverso da zero)?

- (a)  $p == a[0]$ ;
- (b)  $p == \&a[0]$ ;
- (c)  $*p == a[0]$ ;
- (d)  $p[0] == a[0]$ ;

- W 6. Riscrivete la funzione seguente in modo da usare l'aritmetica dei puntatori al posto dell'indicizzazione (in altre parole, eliminare la variabile  $i$  e tutti gli utilizzi dell'operatore  $[ ]$ ). Effettuate il minor numero possibile di modifiche.

```
int sum_array(const int a[], int n)
{
 int i, sum;
 sum = 0;
 for (i = 0; i < n; i++)
 sum += a[i];
 return sum;
}
```

7. Scrivete la seguente funzione

```
bool search(const int a[], int n, int key);
```

dove  $a$  è un vettore nel quale si deve effettuare la ricerca,  $n$  è il numero di elementi del vettore e  $key$  è la chiave di ricerca. La funzione deve restituire `true` se  $key$  combacia con qualche elemento di  $a$ , `false` altrimenti. Per visitare tutti gli elementi del vettore utilizzate l'aritmetica dei puntatori e non l'indicizzazione.

8. Riscrivere la funzione seguente in modo da utilizzare l'aritmetica dei puntatori invece dell'indicizzazione (in altre parole, eliminate la variabile  $i$  e tutti gli utilizzi dell'operatore  $[ ]$ ). Effettuate il minor numero possibile di modifiche.

```
void store_zeros(int a[], int n)
{
 int i;
 for(i = 0, i < n; i++)
 a[i] = 0;
}
```

9. Scrivete la seguente funzione.

```
double inner_product(const double *a, const double *b, int n);
```

$a$  e  $b$  puntano a vettori di lunghezza  $n$ . La funzione deve restituire  $a[0] * b[0] + a[1] * b[1] + \dots + a[n-1] * b[n-1]$ . Per visitare tutti gli elementi dei vettori utilizzate l'aritmetica dei puntatori e non l'indicizzazione.

10. Modificate la funzione `find_middle` della Sezione 11.5 in modo che utilizzi l'aritmetica dei puntatori per calcolare il valore da restituire.

11. Modificate la funzione `find_largest` in modo da utilizzare l'aritmetica dei puntatori (e non l'indicizzazione) per visitare tutti gli elementi del vettore.

12. Scrivete la seguente funzione:

```
void find_two_largest(const int *a, int *n, int *largest, int *second_largest);
```

dove `a` punta a un vettore di lunghezza `n`. La funzione cerca il più grande e il secondo più grande elemento del vettore memorizzandoli rispettivamente nelle variabili puntate da `largest` e `second_largest`. Per visitare tutti gli elementi del vettore usate l'aritmetica dei puntatori e non l'indicizzazione.

#### Sezione 12.4



13. Nella Sezione 8.2 vi è una porzione di programma dove due cicli `for` annidati inizializzano il vettore `ident` al fine di utilizzarlo come matrice identità. Riscrivete quel codice utilizzando un solo puntatore che attraversi tutto il vettore passando per ogni elemento. *Suggerimento:* dato che non useremo le variabili indice `row` e `col`, non sarà facile specificare dove memorizzare gli 1. Possiamo invece sfruttare il fatto che il primo elemento dovrà essere un 1, che dopo `N` elementi ci sarà un altro 1, dopo altri `N` elementi ci sarà un altro 1 e così via. Utilizzate una variabile per tenere traccia di quanti 0 consecutivi avete memorizzato. Quando raggiungete il numero `N` vorrà dire che è tempo di mettere un 1.

14. Assumete che il vettore seguente contenga una settimana di letture orarie della temperatura, dove ogni riga contiene le letture di una giornata:

```
int temperatures[7][24];
```

Scrivete un'istruzione che usi la funzione `search` (guardate l'Esercizio 7) per cercare il valore 32 all'interno dell'intero vettore.

15. Scrivete un ciclo che stampi tutte le letture di temperatura contenute nella riga `i` del vettore `temperatures` (guardate l'Esercizio 14). Utilizzate un puntatore per visitare tutti gli elementi della riga.

16. Scrivete un ciclo che stampi per ogni giorno della settimana la temperatura più alta presente nel vettore `temperatures` (guardate l'Esercizio 14). Il corpo del ciclo dovrà invocare la funzione `find_largest` passandole una riga del vettore per volta.

17. Riscrivete la funzione seguente in modo da usare l'aritmetica dei puntatori invece dell'indicizzazione (in altre parole: eliminate le variabili `i` e `j` e tutti gli usi dell'operatore `[]`). Invece di due cicli annidati utilizzatene uno solo.

```
int sum_two_dimensional_array(const int a[][LEN], int n)
{
 int i, j, sum = 0;
 for(i = 0; i < n; i++)
 for(j = 0; j < LEN; j++)
 sum += a[i][j];
 return sum;
}
```

18. Scrivete la funzione `evaluate_position` descritta nell'Esercizio 13 del Capitolo 9. Per visitare tutti gli elementi del vettore usate l'aritmetica dei puntatori e non l'indicizzazione. In luogo di due cicli annidati utilizzatene uno solo.

## Progetti di programmazione

1. (a) Scrivete un programma che legga un messaggio e che successivamente lo stampi al contrario:

Enter a message: Don't get mad, get even.

Reversal is: .neve teg , dam teg t'noD

*Suggerimento:* Leggete un carattere alla volta (usando la funzione `getchar`) e memorizzate i caratteri in un vettore. Fermatevi quando il vettore è pieno o quando viene letto il carattere '`\n`'.

- (b) Modificate il programma facendo in modo che per tenere traccia della posizione corrente nel vettore venga usato un puntatore invece di un intero.

2. (a) Scrivete un programma che legga un messaggio e poi controlli se questo è palindromo (le lettere del messaggio sono le stesse sia leggendolo da sinistra a destra che da destra a sinistra):

Enter a message: He lived as a devil, eh?

Palindrome

Enter a message: Madam, I am Adam

Not a palindrome

Ignorete tutti i caratteri che non sono lettere. Utilizzare delle variabili intere per tenere traccia delle posizioni all'interno del vettore.

- (b) Modificate il programma in modo da utilizzare dei puntatori invece che degli interi per tenere traccia delle posizioni all'interno del vettore.

- 3. Semplificate il Progetto di programmazione 1(b) sfruttando il fatto che il nome di un vettore può essere usato come un puntatore.
- 4. Semplificate il Progetto di programmazione 2(b) sfruttando il fatto che il nome di un vettore può essere usato come un puntatore.
- 5. Modificate il Progetto di programmazione 14 del Capitolo 8 in modo che utilizzi un puntatore per tenere traccia nel vettore della posizione corrente nella frase.
- 6. Modificate il programma `qsort.c` della Sezione 9.6 in modo che `low`, `high` e `middle` siano dei puntatori agli elementi del vettore invece di interi. La funzione `split` dovrà restituire un puntatore e non un intero.
- 7. Modificate il programma `maxmin.c` della Sezione 11.4 in modo che la funzione `max_min` utilizzi un puntatore invece di un intero per tenere traccia all'interno del vettore della posizione corrente.

# 13 Stringhe

Anche se nei capitoli precedenti abbiamo utilizzato variabili `char` e vettori di valori `char`, manca ancora un modo conveniente per elaborare una serie di caratteri (una *stringa* nella terminologia C). Rimedieremo a questa mancanza nel presente capitolo che tratta sia le stringhe *costanti* (o *letterali*, come vengono chiamate nello standard C) che le stringhe *variabili*, cioè che possono cambiare durante l'esecuzione del programma.

La Sezione 13.1 illustra le regole che governano le stringhe letterali, incluse le regole che incorporano le sequenze di escape nelle stringhe e quelle che spezzano lunghe stringhe letterali. La Sezione 13.2 mostra come dichiarare le stringhe variabili, che sono vettori di caratteri nei quali un carattere speciale (il carattere `null`) segna la fine della stringa. La Sezione 13.3 descrive il modo per leggere e scrivere le stringhe. La Sezione 13.4 mostra come scrivere funzioni che elaborino le stringhe e la Sezione 13.5 tratta alcune funzioni della manipolazione delle stringhe nella libreria del C. La Sezione 13.6 presenta idiom che vengono spesso utilizzati per lavorare con le stringhe. Infine la Sezione 13.7 descrive come creare vettori i cui elementi siano dei puntatori a stringhe di lunghezza diversa. Questa sezione spiega anche come un vettore di quel tipo venga utilizzato dal C per fornire ai programmi informazioni sulla riga di comando.

## 13.1 Stringhe letterali

Una **stringa letterale** è una sequenza di caratteri racchiusa tra doppi apici:

"When you come to a fork in the road, take it."

Abbiamo incontrato per la prima volta le stringhe letterali nel Capitolo 2, infatti appaiono spesso come stringhe di formato nelle chiamate alla `printf` o alla `scanf`.

### Sequenze di escape nelle stringhe letterali

Le stringhe letterali possono contenere le stesse sequenze di escape [sequenze di escape > 7.3] dei costanti carattere. Da tempo stiamo usando caratteri di escape nelle stringhe di formato delle `printf` e delle `scanf`. Per esempio abbiamo visto che ogni carattere `\n` presente nella stringa

"Candy\nIs dandy\nBut liquor\nIs quicker.\n --Ogden Nash\n"

fa sì che il cursore avanzi alla riga successiva:

```
Candy
Is dandy
But liquor
Is quicker.
--Ogden Nash
```

Sebbene nelle stringhe letterali siano ammessi anche gli escape ottali ed esadecimali, questi non sono comuni come gli escape basati su caratteri.



Fate attenzione a quando utilizzate le sequenze di escape ottali ed esadecimali all'interno delle stringhe letterali. Un escape ottale termina dopo tre cifre oppure con il primo carattere non ottale. Per esempio, la stringa "\1234" contiene due caratteri (\123 e 4), mentre la stringa "\189" contiene tre caratteri (\1, 8 e 9). Una sequenza esadecimale d'altra parte non è limitata a tre cifre: non termina fino a quando non incontra il primo carattere non esadecimale. Considerate cosa succederebbe se una stringa contenesse l'escape \xfc che rappresenta il carattere ü nel set di caratteri Latin1 (un'estensione comune del codice ASCII). La stringa "Z\xfcrich" ("Zürich") ha sei caratteri (Z, \xfc, r, i, c e h), mentre la stringa "\xfcber" (un tentativo errato di scrivere "über") ne ha solamente due (\xfcbe ed r). La maggior parte dei compilatori rigetterà l'ultima stringa in quanto gli escape esadecimali di solito sono limitati entro il range \x0-\xff.



## Proseguire una stringa letterale

Se troviamo una stringa letterale che è troppo lunga per essere inserita in modo adeguato su una singola riga, il C ci permette di continuare nella riga successiva a patto che terminiamo la prima riga con il carattere backslash (\). Nessun carattere deve seguire il \ su quella riga, fatta eccezione per il carattere new-line (che è invisibile) posto alla fine:

```
printf("When you come to a fork in the road, take it.\n
--Yogi Berra");
```

In generale il carattere \ può essere usato per unire due o più righe di un programma in modo da formarne una sola (lo standard C si riferisce a questo processo con il nome di *splicing*). Vedremo più esempi di splicing nella Sezione 14.3.

La tecnica del backslash presenta un inconveniente: la stringa deve continuare all'inizio della riga successiva demolendo la struttura indentata del programma. C'è un modo migliore per gestire le stringhe letterali lunghe, derivante dalla seguente regola: quando due o più stringhe letterali sono adiacenti (separate solo da uno spazio bianco), il compilatore le unirà in una singola stringa. Questa regola ci permette di dividere una stringa letterale su due o più righe:

```
printf("When you come to a fork in the road, take it.\n"
"--Yogi Berra");
```

## Come vengono memorizzate le stringhe letterali

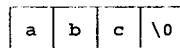
Abbiamo usato spesso le stringhe letterali nelle chiamate alla `printf` e alla `scanf`. Ma quando chiamiamo la `printf` e le passiamo una stringa letterale come argomento, cosa stiamo passando effettivamente? Per rispondere a questa domanda abbiamo bisogno di conoscere come vengono memorizzate le stringhe letterali.

Sostanzialmente il C tratta le stringhe letterali come vettori di caratteri. Quando il compilatore C in un programma incontra una stringa letterale di lunghezza  $n$ , questo alloca per la stringa  $n + 1$  byte di memoria. Quest'area di memoria conterrà i caratteri della stringa con l'aggiunta di un carattere extra (il **carattere null**) per segnare la fine della stringa. Il carattere null è un byte i cui bit sono tutti a zero, e quindi rappresentato dalla sequenza di escape `\0`.



Non confondete il carattere null ('`\0`') con il carattere zero ('`0`'). Il carattere null ha codice zero mentre il carattere zero ha un codice diverso (48 nel codice ASCII).

Per esempio, la stringa letterale "abc" viene memorizzata come un vettore di quattro caratteri (a, b, c e `\0`):



Le stringhe letterali possono essere vuote, la stringa "" viene memorizzata come un singolo carattere null:



Dato che una stringa letterale viene memorizzata come un vettore, il compilatore la tratta come un puntatore di tipo `char *`. Per esempio, sia la `printf` che la `scanf` aspettano un valore del tipo `char *` come loro primo argomento. Considerate l'esempio seguente:

```
printf("abc");
```

Quando la `printf` viene invocata, le viene passato l'indirizzo di "abc" (un puntatore alla locazione di memoria che contiene la lettera a).

## Operazioni sulle stringhe letterali

In generale possiamo usare una stringa letterale ovunque il C ammetta un puntatore di tipo `char *`. Per esempio, una stringa letterale può apparire sul lato destro di un assegnamento.

```
char *p;
```

```
p = "abc";
```

Questo assegnamento non copia i caratteri "abc", semplicemente fa sì che il puntatore p punti alla prima lettera della stringa.

Il C permette ai puntatori di essere indicizzati e di conseguenza possiamo indicizzare anche le stringhe letterali:

```
char ch;
ch = "abc"[1];
```

la lettera b sarà il nuovo valore di ch. Gli altri possibili indici sono lo 0 (che selezionerebbe la lettera a), il 2 (la lettera c) e il 3 (il carattere null). Questa proprietà delle stringhe letterali non è molto utilizzata ma in certe occasioni è comoda. Considerate la seguente funzione che converte un numero compreso tra 0 e 15 in un carattere rappresentante la cifra esadecimale equivalente:

```
char digit_to_hex_char(int digit)
{
 return "0123456789ABCDEF"[digit];
}
```

 Cercare di modificare una stringa letterale provoca un comportamento indefinito:

```
char *p = "abc";
p = 'd'; / SBAGLIATO */
```

 Un programma che cerchi di modificare una stringa letterale potrebbe andare in crash o comportarsi in modo imprevedibile.

## Stringhe letterali e costanti carattere a confronto

Una stringa letterale contenente un singolo carattere non è uguale a una costante carattere. La stringa letterale "a" è rappresentata da un *puntatore* alla locazione di memoria che contiene il carattere a (seguito da un carattere null). La costante carattere 'a' è rappresentata da un *intero* (il codice numerico del carattere).

 Non utilizzate mai un carattere quando viene richiesta una stringa (e viceversa). La chiamata

```
printf("\n");
```

è accettabile perché la printf si aspetta un puntatore come primo argomento. La chiamata seguente invece non è ammissibile:

```
printf('\'\n'); /* SBAGLIATO */
```

## 13.2 Variabili stringa

Alcuni linguaggi di programmazione forniscono uno speciale tipo *string* per dichiarare delle variabili stringa. Il C segue un'altra via: un vettore unidimensionale di caratteri può essere utilizzato per memorizzare una stringa a patto che questa termini con il carattere null. Questo approccio è semplice, ma presenta diverse difficoltà. A volte è difficile capire se un vettore di caratteri è utilizzato come una stringa. Se scri-

viamo nostre funzioni per la manipolazione delle stringhe, dobbiamo fare in modo che queste gestiscano il carattere null in modo appropriato. Inoltre per determinare la lunghezza di una stringa non c'è un metodo più rapido che quello di controllare ogni carattere in modo da trovare il carattere null.

Diciamo che abbiamo bisogno di una variabile che sia capace di contenere una stringa lunga fino a 80 caratteri. Dato che la stringa deve terminare con il carattere null, dichiareremo la variabile come un vettore di 81 caratteri:

```
#define STR_LEN 80
```

```
- char str[STR_LEN +1];
```

Abbiamo definito `STR_LEN` uguale a 80 invece di 81 per enfatizzare il fatto che la str non può contenere più di 80 caratteri. Successivamente abbiamo sommato un 1 a `STR_LEN` all'atto della dichiarazione di `str`. Questa è una pratica molto comune tra i programmati C.



Quando dichiarate un vettore di caratteri che verrà utilizzato per contenere una stringa, a causa della convenzione del C che vuole che tutte le stringhe siano terminate con un carattere null, dovete far sì che il vettore sia più lungo di un carattere rispetto alla stringa che deve contenere. Non lasciare spazio per il carattere null può essere causa di comportamenti impredicibili al momento dell'esecuzione del programma visto che le funzioni della libreria C assumono che le stringhe terminino tutte con il carattere null.

Dichiarare un vettore di caratteri in modo che abbia una lunghezza pari a `STR_LEN` + 1 non significa che questo conterrà sempre una stringa di `STR_LEN` caratteri. La lunghezza di una stringa dipende dalla posizione del carattere di termine e non dalla lunghezza del vettore nel quale è contenuta. Un vettore di `STR_LEN` + 1 caratteri può contenere stringhe di varia lunghezza, che vanno dalla stringa vuota fino a stringhe di lunghezza `STR_LEN`.

## Inizializzare una variabile stringa

Una variabile stringa può essere inizializzata nello stesso momento in cui viene dichiarata.

```
char date1[8] = "June 14";
```

Il compilatore inserirà i caratteri presi da "June 14" nel vettore `date1` e poi aggiungerà il carattere null in modo che il vettore stesso possa essere usato come stringa. Ecco come si presenterà `date1`:

|       |   |   |   |   |  |   |   |    |
|-------|---|---|---|---|--|---|---|----|
| date1 | J | u | n | e |  | 1 | 4 | \0 |
|-------|---|---|---|---|--|---|---|----|

Sebbene "June 14" sembri essere una stringa letterale, non lo è. Il C la vede come un'abbreviazione dell'inizializzatore di un vettore. Infatti avremmo potuto scrivere

```
char date1[8] = {'J', 'u', 'n', 'e', ' ', '1', '4', '\0'};
```

Sarete d'accordo nel convenire che la prima versione sia molto più facile da leggere.

Cosa succederebbe se l'inizializzatore fosse troppo corto per riempire la variabile stringa? In tal caso il compilatore aggiungerebbe caratteri null aggiuntivi. Quindi, dopo la dichiarazione

```
char date2[9] = "June 14";
```

date2 si presenterebbe in questo modo:

|       |   |   |   |   |  |   |   |    |    |
|-------|---|---|---|---|--|---|---|----|----|
| date2 | J | u | n | e |  | 1 | 4 | \0 | \0 |
|-------|---|---|---|---|--|---|---|----|----|

Questo comportamento è coerente con il modo in cui il C generalmente tratta gli inizializzatori dei vettori [[inizializzatori per i vettori > 8.1](#)]. Quando un inizializzatore è più corto del vettore, gli elementi rimanenti vengono inizializzati a zero. Inizializzando con \0 gli elementi rimasti di un vettore di caratteri, il compilatore segue la stessa regola.

Cosa succederebbe se l'inizializzatore fosse più lungo della variabile stringa? Questa situazione non viene ammessa per le stringhe esattamente come non viene ammessa per gli altri vettori. Tuttavia il C permette che l'inizializzatore (senza contare il carattere null) sia esattamente della stessa lunghezza della variabile:

```
char date3[7] = "June 14";
```

Non c'è alcuno spazio per il carattere null e quindi il compilatore non tenta di mettere uno:

|       |   |   |   |   |  |   |   |
|-------|---|---|---|---|--|---|---|
| date3 | J | u | n | e |  | 1 | 4 |
|-------|---|---|---|---|--|---|---|



Se state progettando di inizializzare un vettore di caratteri per contenere una stringa, assicuratevi che la lunghezza del vettore sia maggiore di quella dell'inizializzatore. In caso contrario il compilatore ometterà tranquillamente il carattere null rendendo il vettore non usabile come stringa.

La dichiarazione di una variabile stringa può omettere la sua lunghezza che in tal caso verrà calcolata dal compilatore:

```
char date4[] = "June 14";
```

Il compilatore riserva otto caratteri per il vettore date4, sufficienti per contenere i caratteri presenti in "June 14" assieme al carattere null (il fatto che la lunghezza di date4 non sia specificata non significa che questa possa essere successivamente modificata. Una volta che il programma viene compilato la lunghezza di date4 viene fissata al valore otto). Omettere la lunghezza di una variabile stringa è utile specialmente nei casi in cui l'inizializzatore è lungo, visto che calcolarne a mano la lunghezza è una fonte di errori.

## Vettori di caratteri e puntatori a caratteri a confronto

Confrontiamo la dichiarazione

`char date [] = "June 14";`

la quale dichiara `date` come un *vettore*, con la dichiarazione simile

`char *date4 = "June 14";`

che invece dichiara `date` come un *puntatore*. Grazie alla stretta relazione esistente tra vettori e puntatori possiamo utilizzare entrambe le versioni di `date`. In particolare, qualsiasi funzione che si aspetti che le venga passato un vettore di caratteri o un puntatore a carattere, accetterà come argomento entrambe le versioni di `date`.

Tuttavia, non dobbiamo pensare che le due versioni di `date` siano intercambiabili; tra le due vi sono significative differenze.

- Nella versione vettore, i caratteri che sono presenti in `date` possono essere modificati come gli elementi di un vettore. Nella versione puntatore, `date` punta a una stringa letterale. Nella Sezione 13.1 abbiamo visto che le stringhe letterali non devono essere modificate.
- Nella versione vettore, `date` è il nome di un vettore. Nella versione puntatore `date` è una variabile che può essere fatta puntare ad altre stringhe durante l'esecuzione del programma.

Se abbiamo bisogno di una stringa che possa essere modificata, è nostra responsabilità creare un vettore di caratteri nel quale memorizzare la stringa stessa. Dichiarare una variabile puntatore non è sufficiente. La dichiarazione

`char *p;`

fa sì che il compilatore riservi memoria sufficiente per una variabile puntatore. Sfortunatamente non alloca spazio per una stringa (e come potrebbe? Non abbiamo indicato quanto dovrebbe essere lunga questa stringa). Prima di poter utilizzare la variabile `p` come una stringa dobbiamo farla puntare a un vettore di caratteri. Una possibilità è quella di far puntare `p` a una variabile stringa:

`char str[STR_LEN+1], *p;`

`p = str;`

adesso `p` punta al primo carattere di `str` e quindi possiamo usarla come una stringa. Un'altra possibilità è quella di far puntare `p` a una stringa allocata dinamicamente [**stringhe allocate dinamicamente > 17.2**].

---

 Utilizzare una variabile puntatore non inizializzata come stringa è un errore molto grave. Considerate l'esempio seguente che cerca di formare la stringa "abc":

```
char *p;
p[0] = 'a'; /* SBAGLIATO */
p[1] = 'b'; /* SBAGLIATO */
p[2] = 'c'; /* SBAGLIATO */
p[3] = '\0'; /* SBAGLIATO */
```

Dato che non abbiamo inizializzato la variabile, non sappiamo dove questi punti. Utilizzare la variabile `p` per scrivere in memoria i caratteri `a`, `b`, `c` e `\0` provoca un comportamento indefinito.

### 13.3 Leggere e scrivere le stringhe

Scrivere una stringa è piuttosto facile utilizzando sia la funzione `printf` sia la funzione `puts`. Leggere una stringa è un po' più complicato, principalmente a causa del fatto che la stringa di input può essere più lunga della variabile nella quale deve essere memorizzata. Per leggere una stringa in un colpo solo possiamo usare sia la funzione `scanf` sia la `gets`. Come alternativa possiamo leggere le stringhe un carattere alla volta.

#### Scrivere una stringa con le funzioni `printf` e `puts`

La specifica di conversione `%s` permette alla funzione `printf` di scrivere una stringa. Considerate l'esempio seguente:

```
char str[] = "Are we having fun yet?";
printf("%s\n", str);
```

L'output sarà

`Are we having fun yet?`

La `printf` scrive i caratteri contenuti in una stringa uno alla volta, fino a quando non incontra il carattere null (se il carattere null non è presente, la `printf` continua andando oltre la fine della stringa fin quando, eventualmente, non trova un carattere null da qualche parte nella memoria).

Per stampare solo una parte di una stringa possiamo utilizzare la specifica di conversione `%ps`, dove `p` è il numero di caratteri che devono essere stampati. L'istruzione

```
printf("%.6s\n", str);
```

stamperà

`Are we`

Una stringa, come un numero, può essere stampata all'interno di un campo. La conversione `%ms` visualizzerà una stringa in un campo di dimensione `m` (una stringa con più di `m` caratteri verrà stampata per intero, non verrà troncata). Se una stringa ha meno di `m` caratteri verrà allineata a destra all'interno del campo. Per forzare l'allineamento a sinistra, invece, possiamo mettere un segno meno davanti a `m`. I valori `m` e `p` possono essere usati congiuntamente: una specifica di conversione della forma `%m.ps` fa sì che i primi `p` caratteri della stringa vengano visualizzati in un campo di dimensione `m`.

La funzione `printf` non è l'unica che può scrivere delle stringhe. La libreria C fornisce anche la funzione `puts` che viene usata nel modo seguente:

```
puts(str);
```

la funzione puts ha un solo argomento (la stringa che deve essere stampata). Dopo la stampa della stringa la puts scrive sempre un carattere new-line e quindi avanza alla riga di output successiva.

## Leggere le stringhe con le funzioni scanf e gets

La specifica di conversione %s permette alla scanf di leggere una stringa e di memorizzarla all'interno di un vettore di caratteri:

```
scanf("%s", str);
```

Nella chiamata alla scanf non c'è la necessità di mettere l'operatore & davanti alla variabile str. Come ogni vettore, anche la variabile str viene trattata come un puntatore quando viene passata a una funzione.

Quando viene invocata, la scanf salta gli spazi bianchi e successivamente legge tutti i caratteri salvandoli in str fino a quando non incontra un carattere che rappresenta uno spazio bianco. La scanf mette sempre il carattere null alla fine della stringa.

Una stringa letta usando la funzione scanf non conterrà mai degli spazi bianchi. Quindi, solitamente, la scanf non legge un'intera riga dell'input. Un carattere new-line interrompe la lettura della scanf ma lo stesso effetto viene prodotto anche da uno spazio o da una tabulazione. Per leggere un'intera riga di input in una volta sola possiamo usare la funzione gets. Come la scanf, anche la funzione gets legge i caratteri di input, li immagazzina in un vettore e alla fine aggiunge un carattere null. Tuttavia per altri aspetti la gets possiede delle differenze rispetto alla scanf.

- La gets non salta gli spazi bianchi che precedono l'inizio della stringa (la scanf lo fa).
- La gets legge fino a quando non trova un carattere new-line (la scanf si ferma a qualsiasi carattere che rappresenti uno spazio bianco). Tra l'altro la gets scarta il carattere new-line invece di memorizzarlo all'interno del vettore, al suo posto viene inserito il carattere null.

Per vedere la differenza tra la scanf e la gets prendete in considerazione il seguente estratto di programma:

```
char sentence[SENT_LEN+1];
printf("Enter a sentence:\n");
scanf("%s", sentence);
```

Supponete che dopo il messaggio

Enter a sentence:

l'utente immetta la seguente riga:

To C, or not to C: that is the question.

La scanf memorizzerà la stringa "To" nella variabile sentence. La chiamata successiva alla scanf riprenderà la lettura della riga dallo spazio successivo alla parola To:

Ora supponete di rimpiazzare la scanf con la gets:

```
gets(sentence);
```

Quando l'utente immette lo stesso input di prima, la gets salverà all'interno di sentence la stringa

" To C, or not to C: that is the question."



Quando le funzioni scanf e gets salvano i caratteri all'interno di un vettore, non hanno modo di stabilire quando questo sia pieno. Di conseguenza queste funzioni possono andare a salvare dei caratteri oltre la fine del vettore diventando causa di un comportamento indefinito. La scanf può essere resa sicura utilizzando la specifica di conversione %ns, dove n è un intero che indica il massimo numero di caratteri che devono essere memorizzati. Sfortunatamente la gets è intrinsecamente non sicura, la funzione fgets è un'alternativa decisamente migliore [funzione fgets > 22.5].

## Leggere le stringhe carattere per carattere

Dato che per molte applicazioni sia la scanf che la gets sono rischiose e non sufficientemente flessibili, i programmati C scrivono spesso proprie funzioni di input. Leggendo le stringhe un carattere alla volta, queste funzioni garantiscono un più alto grado di controllo rispetto alle funzioni di input standard.

Se decidiamo di progettare una nostra funzione di input, dobbiamo considerare i seguenti problemi.

- La funzione deve saltare gli spazi bianchi che precedono la stringa prima di memorizzarla?
- Qual è carattere provocherà la fine della lettura da parte della funzione: il carattere new-line, un qualsiasi spazio bianco oppure qualsiasi altro carattere? Tale carattere deve essere memorizzato o scartato?
- Che cosa deve fare la funzione nel caso in cui la stringa sia troppo lunga per essere memorizzata? I caratteri extra devono essere scartati o lasciati per la prossima operazione di input?

Supponete di aver bisogno di una funzione che salti i caratteri di spazio bianco e che fermi la lettura al primo carattere new-line (che non viene memorizzato nella stringa) e scarti i caratteri extra. La funzione dovrebbe avere il seguente prototipo:

```
int read_line(char str[], int n);
```

La variabile str rappresenta il vettore nel quale salvare l'input mentre n rappresenta il massimo numero di caratteri che devono essere letti. Se la riga di input contiene più di n caratteri, la funzione read\_line scarterebbe tutti i caratteri aggiuntivi. La read\_line restituirà il numero di caratteri che ha effettivamente memorizzato in str (un numero qualsiasi compreso tra 0 ed n). Potremmo non aver sempre bisogno del valore restituito dalla funzione, tuttavia non è male averlo a disposizione.

La read\_line consiste principalmente di un ciclo che chiama la funzione getchar [funzione getchar > 7.3] per leggere un carattere e poi memorizzarlo all'interno di str a patto che ci sia spazio a sufficienza per farlo. Il ciclo termina quando viene letto il carattere new-line (per la precisione avremmo bisogno che il ciclo termini anche nel

caso in cui la getchar non riesca a leggere un carattere, tuttavia per ora ignoreremo questa complicazione). Ecco la funzione `read_line` completa:

```
int read_line(char str[], int n)
{
 int ch, i = 0;

 while ((ch = getchar()) != '\n')
 if (i < n)
 str[i++] = ch;
 str[i] = '\0'; /* termina la stringa */
 return i; /* il numero dei caratteri memorizzati */
}
```

Osservate che la variabile `ch` è di tipo `int` e non di tipo `char` perché la funzione `getchar` restituisce un carattere che legge come un valore `int`.

Prima di terminare, la funzione pone un carattere null alla fine della stringa. Funzioni standard come la `scanf` e la `gets` mettono automaticamente un carattere null alla fine della stringa di input, ma se stiamo scrivendo la nostra personale funzione di input, dobbiamo farci carico di questa responsabilità.

## 13.4 Accedere ai caratteri di una stringa

Considerato il fatto che le stringhe vengono memorizzate come dei vettori, possiamo utilizzare l'indicizzazione per accedere ai caratteri contenuti all'interno di queste ultime. Per esempio, per elaborare ogni carattere di una stringa `s`, possiamo creare un ciclo che incrementi il contatore `i` e selezioni i caratteri attraverso l'espressione `s[i]`.

Supponete di aver bisogno di una funzione che conti il numero di spazi presenti in una stringa. Usando l'indicizzazione potremmo scrivere la funzione nel modo seguente:

```
int count_spaces(const char s[])
{
 int count = 0, i;

 for(i = 0; s[i] != '\0'; i++)
 if (s[i] == ' ')
 count++;
 return count;
}
```

Nella dichiarazione di `s` è stata inclusa la parola `const` per indicare che `count_spaces` non modifica il valore rappresentato da `s`. Se `s` non fosse stata una stringa, la funzione avrebbe avuto bisogno di un secondo argomento che specificasse la lunghezza del vettore. Tuttavia, dato che `s` è una stringa, la funzione `count_spaces` può determinare dove questo termina controllando la presenza del carattere null.

Molti programmati C non avrebbero scritto la funzione `count_spaces` in questo modo, ma avrebbero usato un puntatore per tenere traccia della posizione corrente all'interno della stringa. Come abbiamo visto nella Sezione 12.2, questa tecnica è

sempre disponibile per l'elaborazione dei vettori, ma si dimostra estremamente conveniente quando si lavora con le stringhe.

Riscriviamo la funzione `count_spaces` utilizzando l'aritmetica dei puntatori al posto dell'indicizzazione. Elimineremo la variabile `i` e useremo la stessa variabile `s` per tenere traccia della nostra posizione all'interno della stringa. Incrementando `s` ripetutamente, la `count_spaces` può toccare ogni carattere presente nella stringa. Ecco la nostra nuova versione:

```
int count_spaces(const char *s)
{
 int count = 0;

 for(; *s != '\0'; s++)
 if (*s == ' ')
 count++;
 return count;
}
```

Tenete presente che la parola `const` non previene le modifiche di `s` da parte della funzione, ma serve per impedire che la funzione modifichi ciò a cui `s` punta. È dato che `s` è una copia del puntatore che viene passato a `count_spaces`, incrementare `s` non ha effetti sul puntatore originale.

L'esempio `count_spaces` sollecita alcune domande sul modo di scrivere le funzioni per le stringhe.

- **È meglio usare le operazioni sui vettori o le operazioni su puntatori per accedere ai caratteri della stringa?** Siamo liberi di usare quelle che possono essere più comode, possiamo anche combinare i due tipi. Nella seconda versione di `count_spaces`, l'aver trattato `s` come un puntatore semplifica leggermente la funzione rimuovendo la necessità della variabile `i`. Tradizionalmente i programmati C tendono a usare i puntatori per elaborare le stringhe.
- **Un parametro costituito da una stringa deve essere dichiarato come un vettore o come un puntatore?** Le due versioni di `count_spaces` illustrano le opzioni possibili: la prima versione dichiara `s` come un vettore, la seconda dichiara `s` come un puntatore. Effettivamente non c'è differenza tra le due dichiarazioni. Ricordate dalla Sezione 12.3 che il compilatore tratta un parametro costituito da un vettore come se fosse stato dichiarato come puntatore.
- **La forma del parametro (`s[]` o `*s`) ha conseguenze su quello che può essere passato come argomento?** No. Quando la funzione `count_spaces` viene chiamata l'argomento può essere: il nome di un vettore, una variabile puntatore o una stringa letterale. La funzione `count_spaces` non può individuare la differenza.

## 13.5 Usare la libreria C per le stringhe

Alcuni linguaggi di programmazione forniscono operatori che sono in grado di copiare delle stringhe, di confrontarle, di concatenarle, di estrarre da esse delle sottostinghe e cose di questo tipo. Gli operatori C, al contrario, sono essenzialmente inutili per lavorare con le stringhe. Nel C le stringhe vengono trattate come vettori

e quindi subiscono le stesse restrizioni di questi ultimi, in particolare non possono essere copiate o confrontate per mezzo degli operatori.



Tentativi diretti di copiare o confrontare delle stringhe non andranno a buon fine. Per esempio, supponete che str1 e str2 siano state dichiarate in questo modo:

```
char str1[10], str2[10];
```

Non è possibile copiare una stringa in un vettore di caratteri utilizzando l'operatore = :

```
str1 = "abc"; /* SBAGLIATO */
str2 = str1; /* SBAGLIATO */
```

Nella Sezione 12.3 abbiamo visto che non è ammesso l'utilizzo del nome di un vettore come operando sinistro dell'operatore =. È ammissibile invece l'inizializzazione di un vettore di caratteri con l'operatore = :

```
char str1[10] = "abc";
```

Nel contesto di una dichiarazione = non rappresenta l'operatore di assegnamento.

Cercare di confrontare delle stringhe utilizzando un operatore relazionale o di uguaglianza è ammesso, sebbene non produca il risultato desiderato:

```
if (str1 == str2) ... /* SBAGLIATO */
```

Questa istruzione confronta str1 e str2 intesi come *puntatori*, non confronta i contenuti dei due vettori. Dato che str1 e str2 hanno degli indirizzi diversi, l'espressione str1 == str2 dovrà avere il valore 0.

Fortunatamente non tutto è perduto: la libreria del C fornisce un ricco insieme di funzioni adatte a eseguire operazioni sulle stringhe. I prototipi di queste funzioni risiedono nell'header <string.h> [header <string.h> 23.6] e quindi i programmi che necessitano di eseguire operazioni sulle stringhe devono contenere la seguente riga di codice:

```
#include <string.h>
```

La maggior parte delle funzioni dichiarate all'interno di <string.h> richiede almeno una stringa come argomento. I parametri costituiti da una stringa sono del tipo char \* permettendo così che l'argomento possa essere: un vettore di caratteri, una variabile di tipo char \* o una stringa letterale (tutti questi tipi sono accettati come stringhe). Fate attenzione a quei parametri costituiti da una stringa che non sono dichiarati const. Quel tipo di parametri potrebbe essere modificato quando la funzione viene chiamata e quindi l'argomento corrispondente non potrà essere una stringa letterale.

Ci sono diverse funzioni all'interno di <string.h>, noi tratteremo alcune delle più basilari. Negli esempi seguenti assumete che str1 e str2 siano dei vettori di caratteri utilizzati come stringhe.

## La funzione strcpy (string copy)

La funzione strcpy dell'header <string.h> ha il seguente prototipo:

```
char *strcpy(char *s1, const char *s2);
```

In `strcpy` copia la stringa `s2` all'interno della stringa `s1` (per essere precisi dovremmo dire che "la `strcpy` copia la stringa puntata da `s2` nel vettore puntato da `s1`"). Questo significa che la funzione `strcpy` copia in `s1` i caratteri presenti in `s2` fino (e incluso) al primo carattere null che viene incontrato in `s2`. La stringa puntata da `s2` non viene modificata e per questo viene dichiarata `const`.

L'esistenza di questa funzione compensa il fatto di non poter utilizzare l'operatore di assegnamento per copiare delle stringhe. Per esempio, supponete di voler salvare in `str2` la stringa "abcd". Non possiamo usare l'assegnamento

```
str2 = "abcd"; /* *** SBAGLIATO ***/
```

perché `str2` è il nome di un vettore e non può essere utilizzato come membro sinistro dell'operatore di assegnamento. Invece possiamo chiamare la `strcpy`:

```
strcpy(str2, "abcd"); /* adesso str2 contiene "abcd" */
```

Analogamente non ci è permesso assegnare direttamente `str2` a `str1`, ma possiamo invece chiamare la `strcpy`:

```
strcpy(str1, str2); /* adesso str1 contiene "abcd" */
```

La maggior parte delle volte ignoreremo il valore restituito dalla `strcpy`. Occasionalmente però potrebbe essere utile chiamare la `strcpy` come parte di un'espressione più grande in modo da utilizzare il suo valore restituito. Per esempio, possiamo concatenare assieme una serie di chiamate alla `strcpy`:

```
strcpy(str1, strcpy(str2, "abcd"));
/* adesso sia str1 che str2 contengono "abcd" */
```

---

 Nella chiamata `strcpy(str1, str2)`, la funzione `strcpy` non ha modo di verificare che la stringa puntata da `str2` possa essere effettivamente contenuta dal vettore puntato da `str1`. Supponete che `str1` punti a un vettore di lunghezza  $n$ . Se la stringa puntata da `str2` non ha più di  $n - 1$  caratteri allora la copia avrà successo. Se invece `str2` punta a una stringa più lunga, allora si verifica un comportamento indefinito (visto che la `strcpy` copia sempre fino al primo carattere null, la funzione continuerà a copiare anche oltre la fine del vettore puntato da `str1`).

Chiamare la funzione `strncpy` [[funzione `strncpy` > 23.6](#)] è un modo più sicuro, sebbene più lento, di copiare una stringa. La funzione `strncpy` è simile alla `strcpy` ma possiede un terzo argomento che limita il numero di caratteri che verrà copiato. Per copiare `str2` in `str1` possiamo utilizzare la seguente invocazione alla `strncpy`:

```
strncpy(str1, str2, sizeof(str1));
```

Fintanto che `str1` è grande a sufficienza per contenere la stringa memorizzata in `str2` (incluso il carattere null), la copia verrà effettuata correttamente. Tuttavia, la stessa `strcpy` non è priva di pericoli e questo per una ragione: lascerà la stringa in `str1` senza il carattere di terminazione se la lunghezza della stringa contenuta in `str2` è maggiore o uguale alla dimensione del vettore `str1`. Ecco un modo più sicuro di utilizzare la `strncpy`:

```
strncpy(str1, str2, sizeof(str1) - 1);
str1[sizeof(str1)-1] = '\0';
```

La seconda istruzione garantisce che str1 termini sempre con il carattere null, anche quando strncpy non è in grado di copiare il carattere null dalla stringa str2.

## La funzione strlen (string length)

La funzione strlen ha il seguente prototipo:

```
size_t strlen(const char *s);
```

size\_t è un nome typedef definito dalla libreria del C che rappresenta uno dei tipi interi senza segno del C [tipo size\_t > 7.6]. A meno di non lavorare con stringhe estremamente lunghe, questo tecnicismo non ci deve preoccupare, possiamo semplicemente trattare il valore restituito dalla strlen come un intero.

La strlen restituisce la lunghezza della stringa s, ovvero il numero di caratteri presenti in s fino al primo carattere null, quest'ultimo escluso. Ecco alcuni esempi:

```
int len;
len = strlen("abc"); /* adesso len è uguale a 3 */
len = strlen(""); /* adesso len è uguale a 0 */
strcpy(str1, "abc");
len = strlen(str1); /* adesso len è uguale a 3 */
```

L'ultimo esempio illustra un punto molto importante. Quando alla strlen viene passato un vettore come argomento, questa non misura la lunghezza del vettore, ma la lunghezza della stringa in esso contenuta.

## La funzione strcat (string concatenation)

La funzione strcat ha il seguente prototipo:

```
char *strcat(char *s1, const char *s2);
```

la strcat aggiunge il contenuto della stringa s2 alla fine della stringa s1 e restituisce s1 (un puntatore alla stringa risultante).

Ecco alcuni esempi della strcat in azione:

```
strcpy(str1, "abc");
strcat(str1, "def"); /* adesso str1 contiene "abcdef" */
strcpy(str1, "abc");
strcpy(str2, "def");
strcpy(str1, str2); /* Copia abc in str1
 def in str2
 str1 contiene lo stesso.
 str2 contiene "def" */
```

Come accade per la strcpy, anche per la strcat è usuale scartarne il valore restituito. Gli esempi seguenti illustrano come il valore restituito potrebbe essere utilizzato:

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, strcat(str2, "ghi"));
/* adesso str1 contiene "abcdefghi" e str2 contiene "defghi" */
```



L'effetto della chiamata `strcat(str1, str2)` non è definito nel caso in cui il vettore puntato da `str1` non sia lungo a sufficienza per contenere i caratteri aggiuntivi provenienti da `str2`. Considerate l'esempio seguente:

```
char str1[6] = "abc";
strcat(str1, "def"); //*** SBAGLIATO ***/
```

If vettore non è lungo →  
... abc def

la `strcat` cercherebbe di aggiungere i caratteri d, e, f e \0 alla fine della stringa già contenuta in `str1`. Sfortunatamente `str1` è limitata a sei caratteri e a causa di questo la `strcat` andrà a scrivere oltre la fine del vettore.

La funzione `strncat` [funzione `strncat` > 23.6] è una versione della `strcat` più sicura ma più lenta. Come la `strncpy`, ha un terzo argomento che pone un limite al numero di caratteri che verranno copiati. Ecco come potrebbe presentarsi una chiamata alla `strncat`:

```
strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
```

la `strncat` terminerà la stringa `str1` con un carattere null il quale non è incluso nel terzo argomento (il numero di caratteri da copiare). Nell'esempio il terzo argomento calcola il quantitativo di spazio rimanente in `str1` (dato dall'espressione `sizeof(str1) - strlen(str1)`) e poi gli sottrae 1 per assicurarsi che ci sia spazio per il carattere null.

## La funzione `strcmp` (string comparison)

La funzione `strcmp` ha il seguente prototipo:

```
int strcmp(const char *s1, const char *s2);
```

D&R  
La `strcmp` confronta le due stringhe `s1` ed `s2` restituendo un valore minore, uguale o maggiore a 0 a seconda che `s1` sia minore, uguale o maggiore di `s2`. Per esempio, per vedere se `str1` è minore di `str2` scriveremo

```
if (strcmp(str1, str2) < 0) /* str1 < str2 ? */
```

Per esempio, per vedere se `str1` è minore o uguale a `str2` scriveremo

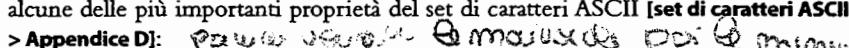
```
if (strcmp(str1, str2) <= 0) /* str1 <= str2 ? */
```

Scegliendo l'operatore relazionale (`<`, `<=`, `>`, `>=`) o di uguaglianza (`==`, `!=`) appropriato, possiamo analizzare tutte le possibili relazioni esistenti tra `str1` e `str2`.

La `strcmp` confronta le stringhe basandosi sul loro ordine lessicografico, che corrisponde all'ordine nel quale le parole vengono sistematiche in un dizionario. Più precisamente, `strcmp` considera `s1` minore di `s2` nel caso in cui una delle seguenti condizioni siano soddisfatte:

- i primi  $i$  caratteri di `s1` ed `s2` combaciano ma l' $(i+1)$ -esimo carattere di `s1` è minore dell' $(i+1)$ -esimo carattere di `s2`. Per esempio, "abc" è minore di "bcd" e "abd" è minore di "abe";

- tutti i caratteri di s1 combaciano con quelli di s2, ma s1 è più corta di s2. Per esempio, "abc" è minore di "abcd".

Quando confronta i caratteri delle due stringhe, la funzione strcmp guarda i codici numerici che rappresentano i caratteri. Una conoscenza del set di caratteri sottostante è utile per capire quale sarà il comportamento di strcmp. Per esempio, queste sono alcune delle più importanti proprietà del set di caratteri ASCII [**set di caratteri ASCII > Appendice D**: 

- i caratteri in ognuna delle sequenze A-Z, a-z e 0-9 hanno codici consecutivi;
- tutte le lettere maiuscole sono minori di quelle minuscole (in ASCII i codici compresi tra 65 e 90 rappresentano le lettere maiuscole, mentre i codici compresi tra 97 e 122 rappresentano le lettere minuscole);
- le cifre hanno codici minori delle lettere (i codici compresi tra 48 e 57 rappresentano le cifre);
- gli spazi hanno codici minori a quelli di tutti i caratteri stampabili (il carattere spazio in ASCII ha codice 32).

## PROGRAMMA

### Stampare i promemoria di un mese

Per illustrare l'utilizzo della libreria C per le stringhe, sviluppiamo un programma che stampi l'elenco dei promemoria giornalieri di un mese. L'utente immette una serie di note, ognuna associata a un giorno del mese. Quando l'utente immette uno 0 invece di un giorno valido, il programma stampa la lista di tutti i promemoria immessi, ordinati per giorno. Ecco come potrebbe presentarsi una sessione del programma:

```

Enter day and reminder: 24 Susan's birthday
Enter day and reminder: 5 6:00 - Dinner with Marge and Russ
Enter day and reminder: 26 movie - "Chinatown"
Enter day and reminder: 7 10:30 - Dental appointment
Enter day and reminder: 12 Movie - "Dazed and Confused"
Enter day and reminder: 5 Saturday class
Enter day and reminder: 12 Saturday class
Enter day and reminder: 0
Day Reminder
5 Saturday class
5 6:00 - Dinner with Marge and Russ
7 10:30 - Dental appointment
12 Saturday class
12 Movie - "Dazed and Confused"
24 Susan's birthday
26 movie - "Chinatown"

```

La strategia complessiva non è molto complicata: il programma leggerà una serie di combinazioni giorno-promemoria, le salverà in ordine (ordinate per giorno) e successivamente le visualizzerà. Per leggere i giorni useremo la funzione scanf mentre per leggere i promemoria useremo la funzione read\_line della Sezione 13.3.

Salveremo le stringhe in un vettore di caratteri bidimensionale dove ogni riga conterrà una stringa. Dopo che il programma avrà letto il giorno e il promemoria associato, cercherà nel vettore dove posizionare il giorno in questione utilizzando la `strcmp` per effettuare i confronti. Successivamente utilizzerà la funzione `strip` per spostare tutte le stringhe al di sotto di quel punto di una posizione in meno. Infine, il programma copierà il giorno nel vettore e invocherà la `strcat` per aggiungervi il promemoria del giorno (il giorno e il promemoria erano stati mantenuti separati fino a questo punto).

Naturalmente ci sono sempre delle complicazioni minori. Per esempio vogliamo che i giorni vengano allineati a destra in un campo di due caratteri in modo che le loro cifre siano allineate. Ci sono molti modi per gestire questo problema. La scelta fatta qui è quella di utilizzare la `scanf` [funzione `scanf` > 22.8] per leggere il giorno e memorizzarlo in una variabile intera. Successivamente viene effettuata una chiamata alla funzione `sprintf` per convertire nuovamente il giorno nel formato stringa. La `sprintf` è una funzione di libreria che è simile alla `printf` ad eccezione del fatto che scrive il suo output in una stringa. La chiamata `sprintf(day_str, "%2d", day);`

scrive il valore della variabile `day` in `day_str`. Dato che la `sprintf` quando scrive aggiunge automaticamente un carattere null, `day_str` verrà a contenere una stringa che termina appropriatamente con un carattere null.

Un'altra complicazione è quella di assicurarsi che l'utente non immetta più di due cifre. A questo scopo utilizzeremo la chiamata seguente:

`scanf("%2d", &day);`

Il numero 2 tra % e d dice alla `scanf` di interrompere la lettura dopo due cifre, anche se l'input ha più caratteri.

Con questi dettagli sistemati, ecco il programma:

```
/* Stampa la lista di promemoria di un mese */
#include <stdio.h>
#include <string.h>

#define MAX_REMIND 50 /* numero massimo di promemoria */
#define MSG_LEN 60 /* lunghezza massima dei messaggi */

int read_line(char str[], int n);

int main(void)
{
 char reminders[MAX_REMIND][MSG_LEN+3];
 char day_str[3], msg_str[MSG_LEN+1];
 int day, i, j, num_remind = 0;

 for (;;) {
 if (num_remind == MAX_REMIND) {
 printf("-- No space left --\n");
 break;
 }
 }
}
```

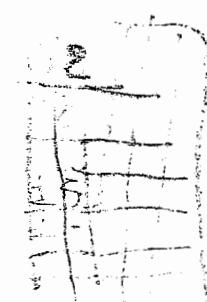
```

printf("Enter day and reminder: ");
scanf("%2d", &day);
if (day == 0)
 break;
sprintf(day_str, "%2d", day);
read_line(msg_str, MSG_LEN);
for (i = 0; i < num_remind; i++)
 if (strcmp(day_str, reminders[i]) < 0)
 break;
for (j = num_remind; j > i; j--)
 strcpy(reminders[j], reminders[j-1]);
strcpy(reminders[i], day_str);
strcat(reminders[i], msg_str);
num_remind++;
}
printf("\nDay' Reminder\n");
for (i = 0; i < num_remind; i++)
 printf(" %s\n", reminders[i]);
return 0;
}

int read_line(char str[], int n)
{
 int ch, i = 0;

 while ((ch = getchar()) != '\n')
 if (i < n)
 str[i] = ch;
 str[i] = '\0';
 return i;
}

```



Sebbene `remind.c` sia utile per dimostrare l'uso delle funzioni `strip`, `strcat` e `strcmp` soffre di alcune mancanze per essere un programma di promemoria usabile. C'è la necessità di un buon numero di perfezionamenti che vanno da alcune piccole messe a punto a miglioramenti più consistenti (come salvare i promemoria in un file quando il programma termina). Discuteremo diversi miglioramenti nei progetti di programmazione alla fine del capitolo e in quelli prossimi.

## 13.6 Idiomi per le stringhe

Le funzioni per manipolare le stringhe sono una fonte di idiomi particolarmente ricca. In questa sezione esploreremo alcuni dei più famosi idiomi usando per scrivere le funzioni `strlen` e `strcat`. Naturalmente non avremo mai bisogno di scrivere queste funzioni dato che fanno parte della libreria standard ma potremmo dover scrivere funzioni che sono simili.

Lo stile conciso che useremo in questa sezione è popolare tra molti programmati C; dovreste padroneggiarlo anche se non progettate di usarlo nei vostri programmi ma è probabile che lo incontriate nel codice scritto da altri.

Un'ultima nota prima di cominciare. Se volete provare una qualsiasi delle versioni di `strlen` e `strcat` di questa sezione assicuratevi di modificare il nome della funzione (cambiando `strlen` in `my_strlen` per esempio). Come spiega la Sezione 21.1 non è consentito scrivere una funzione che abbia lo stesso nome di una funzione della libreria standard anche quando non includiamo l'header al quale appartiene la funzione. Infatti tutti i nomi che iniziano per `str` e una lettera minuscola sono riservati (per permettere l'aggiunta di funzioni all'header `<string.h>` in versioni future dello standard C).

## Cercare la fine di una stringa

Molte delle operazioni sulle stringhe richiedono la ricerca della fine della stringa. La funzione `strlen` ne è un primo esempio. La seguente versione di `strlen` cerca la fine della stringa che rappresenta il suo argomento utilizzando una variabile per tenere traccia della lunghezza della stringa:

```
size_t strlen(const char *s)
{
 size_t n;
 for (n=0; *s != '\0'; s++)
 n++;
 return n;
}
```

Mentre il puntatore `s` si sposta lungo la stringa da sinistra a destra, la variabile `n` tiene traccia di quanti caratteri sono stati visti fino a quel momento. Quando `s` finalmente punta a un carattere null, `n` contiene la lunghezza della stringa.

Vediamo se è possibile condensare la funzione. Per prima cosa sposteremo l'inizializzazione di `n` nella sua dichiarazione:

```
size_t strlen(const char *s)
{
 size_t n = 0;
 for (; *s != '\0'; s++)
 n++;
 return n;
}
```

Successivamente notiamo che la condizione `*s != '\0'` è equivalente `*s != 0` perché il valore intero del carattere null è 0. Ma testare `*s != 0` è equivalente a testare `*s`, entrambe le espressioni sono vere quando `*s` è diverso da zero. Queste osservazioni ci conducono alla nostra versione di `strlen`:

```
size_t strlen(const char *s)
{
 size_t n = 0;
 for (; *s; s++)
 n++;
 return n;
}
```

Nella Sezione 12.2 abbiamo visto che è possibile incrementare e testare `*s` all'interno della stessa espressione:

```
size_t strlen(const char *s)
{
 size_t n = 0;
 for (; *s++);
 n++;
 return n;
}
```

Rimpiazzando l'istruzione `for` con l'istruzione `while` giungiamo alla seguente versione di `strlen`:

```
size_t strlen(const char *s)
{
 size_t n = 0;
 while (*s++)
 n++;
 return n;
}
```

Sebbene abbiano condensato un po' il codice della `strlen`, probabilmente non abbiamo incrementato la sua velocità. Ecco una versione che è più veloce, almeno con alcuni compilatori:

```
size_t strlen(const char *s)
{
 const char *p = s;
 while (*s)
 s++;
 return s - p;
}
```

Questa versione della `strlen` calcola la lunghezza della stringa localizzando la posizione del carattere `null` e poi sottraendo da questa la posizione del primo carattere presente nella stringa. L'incremento della velocità deriva dal non aver incrementato `n` all'interno del ciclo `while`. Osservate l'occorrenza della parola `const` nella dichiarazione di `p`: senza di essa il compilatore noterebbe che assegnando `s` a `p` si porrebbe a rischio la stringa puntata da `s`.

L'istruzione

```
while (*s)
 s++;
```

e la collegata

```
while (*s++)
 ;
```

rappresentano degli idiom che significano "cerca il carattere null alla fine della stringa". La prima versione fa sì che s punti al carattere null. La seconda versione è più concisa ma fa in modo che s punti dopo il carattere null.

## Copiare una stringa

Copiare una stringa è un'altra operazione molto comune. Per introdurre l'idioma C per la copia delle stringhe svilupperemo due versioni della funzione strcat. Iniziamo con una versione immediata ma in qualche modo lunga:

```
char *strcat(char *s1, const char *s2)
{
 char *p = s1;

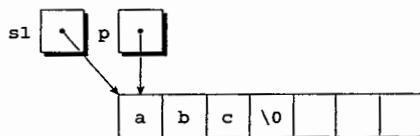
 while (*p != '\0')
 p++;

 while (*s2 != '\0') {
 *p = *s2;
 p++;
 s2++;
 }

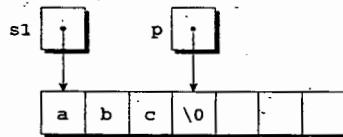
 *p = '\0';
 return s1;
}
```

Questa versione di strcat utilizza un algoritmo a due passi: (1) individua il carattere null alla fine della stringa s1 e fa in modo che p punti a esso; (2) copia i caratteri uno alla volta da s2 nella locazione puntata da p.

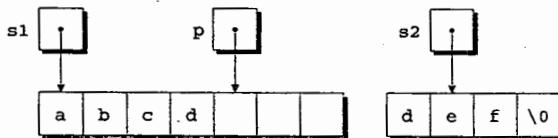
La prima istruzione while della funzione implementa il passo (1). La variabile p viene fatta puntare al primo carattere della stringa s1. Assumendo che s1 punti alla stringa "abc", abbiamo la seguente situazione:



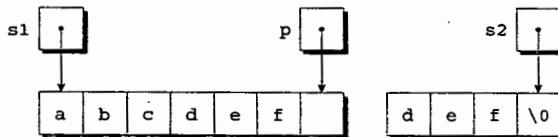
Successivamente p viene incrementata fino a quando non punta a un carattere null. Quando il ciclo termina, p deve puntare al carattere null:



Il secondo ciclo while implementa il passo (2) dell'algoritmo. Il corpo del ciclo copia un carattere dalla locazione puntata da `s2` in quella puntata da `p` e successivamente incrementa sia `p` che `s2`. Se originariamente `s2` puntava alla stringa "def", ecco come appariranno le stringhe dopo la prima iterazione del ciclo:



Il ciclo termina quando `s2` punta al carattere null:



Dopo aver posto il carattere null nella locazione puntata da `p`, la funzione `strcat` termina.

Con un procedimento simile a quello utilizzato per la `strlen`, possiamo riassumere la definizione di `strcat` giungendo alla seguente versione:

```
char *strcat(char *s1, const char *s2)
{
 char *p = s1; P punta alla posizione iniziale della stringa s1
 while (*p) Finché non si incontra il carattere null
 p++;
 while (*p++ = *s2++) Copia il carattere s2 all'indirizzo corrente di p
 ;
 return s1; Se tutto è finito, ritorna s1
}
```

Il cuore della versione snella della funzione `strcat` è l'idioma di "copia della stringa":

```
while (*p++ = *s2++)
 ;
```

Se ignoriamo i due operatori `++`, l'espressione dentro la parentesi si semplifica e otteniamo un normale assegnamento:

```
*p = *s2;
```

Questa espressione copia un carattere dalla locazione puntata da `s2` in quella puntata da `p`. Dopo l'assegnamento sia `p` che `s2` vengono incrementate grazie agli operatori `++`. Eseguire ripetutamente questa espressione ha l'effetto di copiare una serie di caratteri dalle locazioni puntate da `s2` alle locazioni puntate da `p`.

Ma cosa fa concludere il ciclo? Dato che l'operatore primario dentro le parentesi è un assegnamento, l'istruzione `while` analizza il valore di questo, ovvero il carattere che è stato copiato. Tutti i caratteri ad eccezione del carattere null equivalgono alla condizione true e quindi il ciclo non si fermerà fino a quando non è stato copiato il carattere null. Visto che il ciclo termina *dopo* l'assegnamento, non abbiamo bisogno di un'istruzione separata per mettere un carattere null alla fine della nuova stringa.

## 13.7 Vettori di stringhe

Torniamo adesso su un problema che abbiamo incontrato spesso: qual è il modo migliore per memorizzare un vettore di stringhe? La soluzione ovvia è quella di creare un vettore bidimensionale di caratteri e poi mettere le stringhe all'interno del vettore, una per riga. Considerate l'esempio seguente:

```
char planets[][][8] = {"Mercury", "Venus", "Earth",
 "Mars", "Jupiter", "Saturn",
 "Uranus", "Neptune", "Pluto"};
```

(Nel 2006 l'Unione Internazionale di Astronomia ha declassato Plutone da “pianeta” a “pianeta nano”, tuttavia è stato lasciato nel vettore dei pianeti in ricordo dei vecchi tempi). Osservate che ci è permesso di omettere il numero di righe del vettore `planets` visto che questo è determinabile in modo ovvio dal numero di elementi presenti nell'inizializzatore, mentre il C richiede che venga specificato il numero di colonne.

La figura a pagina seguente illustra come apparirà il vettore `planets`. Non tutte le stringhe sono lunghe a sufficienza per occupare un'intera riga del vettore e quindi il C le riempie queste ultime con caratteri null. In questo vettore c'è uno spreco di spazio dato che solo tre pianeti hanno nomi lunghi a sufficienza da richiedere otto caratteri (incluso il carattere di termine). Il programma `remind.c` (Sezione 13.5) è un fulgido esempio di questo tipo di spreco: memorizza i promemoria nelle righe di un vettore bidimensionale, con 60 caratteri riservati per ognuno di essi. Nel nostro esempio i promemoria avevano una lunghezza compresa tra 18 e 37 caratteri, quindi la quantità di spazio sprecato era considerevole.

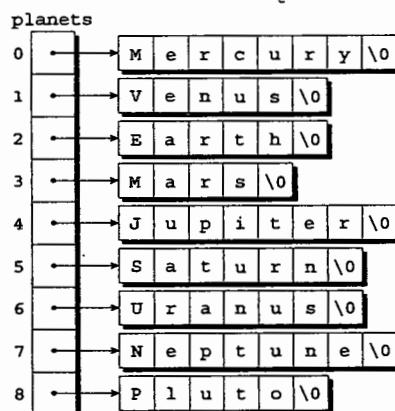
L'inefficienza che appare in questi esempi è comune quando si lavora con le stringhe dato che molte di queste saranno un mix di stringhe lunghe e stringhe corte. Quello di cui abbiamo bisogno è un vettore frastagliato (*rugged array*): un vettore bidimensionale le cui righe hanno lunghezza diversa. Il C non fornisce questo tipo di vettori, tuttavia ci dà un modo per simularli. Il segreto è quello di creare un vettore i cui elementi siano dei *puntatori* a stringhe.

|   | 0 | 1 | 2 | 3 | 4  | 5  | 6  | 7  |
|---|---|---|---|---|----|----|----|----|
| 0 | M | e | r | c | u  | r  | y  | \0 |
| 1 | V | e | n | u | s  | \0 | \0 | \0 |
| 2 | E | a | r | t | h  | \0 | \0 | \0 |
| 3 | M | a | r | s | \0 | \0 | \0 | \0 |
| 4 | J | u | p | i | t  | e  | r  | \0 |
| 5 | S | a | t | u | r  | n  | \0 | \0 |
| 6 | U | r | a | n | u  | s  | \0 | \0 |
| 7 | N | e | p | t | u  | n  | e  | \0 |
| 8 | P | l | u | t | o  | \0 | \0 | \0 |

Ecco di nuovo il vettore planets, creato questa volta come un vettore di puntatori a stringa:

```
char *planets[] = {"Mercury", "Venus", "Earth",
 "Mars", "Jupiter", "Saturn",
 "Uranus", "Neptune", "Pluto"};
```

Non è una grande modifica. Abbiamo semplicemente rimosso un paio di parentesi e messo un asterisco davanti al nome planets. L'effetto sul modo in cui viene memorizzato il vettore però è sostanziale:



Ogni elemento di planets è un puntatore a una stringa terminante con null. Nelle stringhe non ci sono più sprechi di caratteri, sebbene ora abbiamo dovuto allocare dello spazio per i puntatori nel vettore planets.

Per accedere a uno dei nomi dei pianeti, tutto quello di cui abbiamo bisogno è di indicizzare il vettore planets. A causa della relazione tra i puntatori e i vettori, accedere ai caratteri appartenenti al nome di un pianeta viene fatto allo stesso modo nel quale si

accede a un elemento di un vettore bidimensionale. Per cercare nel vettore stringhe che iniziano con la lettera M, per esempio, possiamo utilizzare il seguente ciclo:

```
for (i = 0; i < 9; i++)
 if (planets[i][0] == 'M')
 printf("%s begins with M\n", planets[i]);
```

## Argomenti della riga di comando

Quando eseguiamo un programma capita spesso di aver bisogno di fornirgli delle informazioni, per esempio il nome di un file o una qualche informazione che modifica il comportamento del programma stesso. Considerate il comando UNIX ls. Se eseguiamo ls scrivendo nella riga di comando

ls

questo visualizzerà i nomi dei file presenti nella cartella corrente. Se invece digitiamo

ls -l

allora il programma ls visualizzerà una "lunga" e dettagliata lista di file mostrando la dimensione di ognuno di questi, il proprietario, la data e l'ora della loro ultima modifica e così via. Per modificare ulteriormente il comportamento di ls possiamo chiedergli di mostrare i dettagli di un solo file:

ls -l remind.c

In questo modo ls visualizzerà informazioni dettagliate riguardo il file chiamato remind.c.

**D&R** Le informazioni della riga di comando sono disponibili a tutti i programmi e non solo ai comandi del sistema operativo. Per ottenere accesso a questi **argomenti della riga di comando** (chiamati **parametri del programma** nello standard C), dobbiamo definire il main come una funzione con due parametri che, per consuetudine, vengono chiamati argc e argv:

```
int main(int argc, char *argv[])
{
 ...
}
```

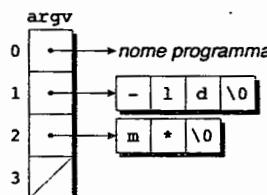
argc (argument count) è il numero di argomenti della riga di comando (incluso il nome dello programma stesso). argv (argument vector) è un vettore di puntatori agli argomenti della riga di comando che sono memorizzati sotto forma di stringhe. L'elemento argv[0] punta al nome del programma, mentre gli elementi da argv[1] ad argv[argc-1] puntano ai restanti argomenti della riga di comando.

Il vettore argv possiede un elemento aggiuntivo, argv[argc], che è sempre un puntatore nullo (null pointer), ovvero uno speciale puntatore che non punta a nulla. Discuteremo dei puntatori nulli più avanti [**puntatori nulli > 17.1**], per ora tutto ciò che ci serve sapére è che la macro NULL rappresenta un puntatore nullo.

Se l'utente immettesse la riga di comando

ls -l remind.c

allora argc sarebbe uguale a 3, argv[0] punterebbe a una stringa costante contenente il nome del programma, argv[1] punterebbe alla stringa "-l", argv[2] punterebbe alla stringa "remind.c", e argv[3] sarebbe un puntatore nullo:



Questa immagine non mostra il nome del programma nel dettaglio perché esso può contenere il percorso (*path*) o altre informazioni che dipendono dal sistema operativo. Se il nome del programma non è disponibile, allora argv[0] punta a una stringa vuota.

Dato che argv è un vettore di puntatori, accedere agli argomenti della riga di comando è piuttosto facile. Tipicamente un programma che si aspetta degli argomenti dalla riga di comando crea un ciclo che esamina tutti gli argomenti uno a uno. Un modo per scrivere questo ciclo è quello di utilizzare una variabile intera come indice per il vettore argv. Per esempio, il ciclo seguente stampa gli argomenti della riga di comando:

```
int i;
for (i = 1; i < argc; i++)
 printf("%s\n", argv[i]);
```

Un'altra tecnica è quella di creare un puntatore ad argv[1], successivamente incrementare ripetutamente il puntatore per toccare tutti gli elementi del vettore. Considerando che l'ultimo elemento di argv è sempre un puntatore nullo, il ciclo può terminare quando incontra un puntatore nullo nel vettore:

```
char **p;
for (p = &argv[1]; *p != NULL; p++)
 printf("%s\n", *p);
```

Dato che p è un puntatore a un puntatore a carattere, dobbiamo utilizzarlo con attenzione. Imporre p uguale a &argv[1] è sensato: argv[1] è un puntatore a un carattere e quindi &argv[1] è un puntatore a un puntatore. Il confronto \*p != NULL è corretto perché sia \*p che NULL sono puntatori. Anche incrementare p è corretto: p punta all'elemento di un vettore e quindi incrementarlo lo farà avanzare al prossimo elemento. La stampa di \*p è corretta perché quest'ultimo punta al primo carattere di una stringa.

## Controllare i nomi dei pianeti

Il nostro prossimo programma, planet.c, illustra come accedere agli argomenti della riga di comando. Il programma è pensato per controllare una serie di stringhe per vedere se ognuna di queste corrisponde al nome di un pianeta. Quando il programma viene eseguito, l'utente deve inserire le stringhe che devono essere testate nella riga di comando:

```
planet Jupiter venus Earth fred
```

Il programma indicherà se ogni stringa è, o non è, il nome di un pianeta. Nel caso in cui la stringa fosse il nome di un pianeta il programma visualizzerà il numero di tale pianeta (assegnando il numero 1 al pianeta più vicino al sole):

```
Jupiter is planet 5
venus is not a planet
Earth is planet 3
fred is not a planet
```

Osservate che il programma non riconosce come nome di pianeta una stringa che non abbia la prima lettera maiuscola e le restanti lettere minuscole.

```
planet.c /* Controlla i nomi dei pianeti */

#include <stdio.h>
#include <string.h>

#define NUM_PLANETS 9

int main(int argc, char *argv[])
{
 char *planets[] = {"Mercury", "Venus", "Earth",
 "Mars", "Jupiter", "Saturn",
 "Uranus", "Neptune", "Pluto"};
 int i, j;

 for (i = 1; i < argc; i++) {
 for (j = 0; j < NUM_PLANETS; j++)
 if (strcmp(argv[i], planets[j]) == 0) {
 printf("%s is planet %d\n", argv[i], j + 1);
 break;
 }
 if (j == NUM_PLANETS)
 printf("%s is not a planet\n", argv[i]);
 }
 return 0;
}
```

Il programma prende un argomento alla volta dalla riga di comando e lo confronta con le stringhe presenti nel vettore planets fino a quando non trova una corrispondenza o giunge alla fine del vettore. La parte più interessante è la chiamata alla funzione strcmp dove gli argomenti sono argv[1] (un puntatore all'argomento della riga di comando) e planets[j] (un puntatore al nome di un pianeta).

## Domande & Risposte

**D: Quanto può essere lunga una stringa letterale?**

**R:** Secondo lo standard C89, i compilatori devono ammettere stringhe letterali lunghe almeno 509 caratteri (non chiedete perché proprio 509). Il C99 ha innalzato questo livello minimo a 4095 caratteri.

**D:** Perché le stringhe letterali non vengono chiamate "stringhe costanti"?

**R:** Perché non sono necessariamente costanti. Dato che le stringhe letterali sono accessibili attraverso i puntatori, non c'è nulla che impedisca al programma di modificare i caratteri presenti in esse.

**D:** Come possiamo scrivere la stringa letterale "über" nel caso in cui "\xfc-ber" non funzionasse? [p. 290]

**R:** Il segreto sta nello scrivere due stringhe letterali adiacenti e lasciare che il compilatore le fonda assieme. Scrivere "\xfc" "ber" ci fornirà una stringa letterale rappresentante la parola "über".

**D:** Modificare una stringa letterale sembra abbastanza inoffensivo. Perché causa un comportamento indefinito? [p. 292]

**R:** Alcuni compilatori cercano di ridurre l'occupazione della memoria memorizzando una sola copia per stringhe letterali identiche. Considerate l'esempio seguente:

```
char *p = "abc", *q = "abc";
```

Un compilatore potrebbe decidere di memorizzare la stringa "abc" una sola volta facendo puntare a questa sia p che q. Se modifichiamo "abc" attraverso il puntatore p, allora anche la stringa puntata da q ne risente. Non c'è bisogno di dire che questo può condurre a bachi piuttosto fastidiosi. Un altro possibile problema è dato dal fatto che le stringhe letterali possono essere memorizzate in un'area della memoria a "sola lettura". Un programma che cercasse di modificare una stringa letterale di quel tipo andrebbe in crash.

**D:** Ogni vettore di caratteri deve includere dello spazio per il carattere null?

**R:** Non necessariamente dato che non tutti i vettori di caratteri vengono usati come delle stringhe. Includere dello spazio per il carattere null (e metterne effettivamente uno all'interno del vettore) è necessario solo se state progettando di passare il vettore a una funzione che richiede delle stringhe terminate con null.

Non avete bisogno del carattere null nel caso in cui stiate eseguendo delle operazioni solo sui singoli caratteri. Per esempio: un programma può possedere un vettore di caratteri utilizzato per effettuare delle traduzioni da un set di caratteri a un altro:

```
char translation_table[128];
```

L'unica operazione che verrà effettuata dal programma sarà l'indicizzazione (il valore di `translation_table[ch]` sarà la versione tradotta del carattere `ch`). Non considereremo `translation_table` come una stringa: nessuna operazione sulle stringhe verrà applicata su di essa e quindi non c'è bisogno che contenga il carattere null.

**D:** Dato che le funzioni `printf` e `scanf` richiedono che il loro primo argomento sia di tipo `char *`, questo significa che tale argomento può essere una variabile stringa invece che una stringa letterale?

**R:** Certamente, proprio come potete vedere nell'esempio seguente:

```
char fmt[] = "%d\n";
int i;
...
printf(fmt, i);
```

Questa abilità apre la porta ad alcune interessanti possibilità (come leggere dall'input una stringa di formato, per esempio).

**D:** Se volessimo stampare la stringa str con la printf, potremmo semplicemente fornire str come stringa di formato come succede nell'esempio seguente?

```
printf(str);
```

**R:** Sì, ma è rischioso. Se str contenesse il carattere % non otterreste il risultato desiderato dato che la printf lo interpreterebbe come l'inizio di una specifica di conversione.

\***D:** Come può fare la funzione read\_line per determinare se la getchar ha fallito nella lettura di un carattere? [p. 298]

**R:** Se non può leggere un carattere a causa di un errore o perché ha incontrato un end-of-file, la getchar restituisce il valore EOF [macro EOF > 22.4] che è di tipo int. Ecco una versione rivisitata di read\_line che controlla se il valore restituito dalla getchar è pari a EOF. Le modifiche sono indicate in **grassetto**:

```
int read_line(char str[], int n)
{
 int ch, i = 0;

 while ((ch = getchar()) != '\n' && ch != EOF)
 if (i < n)
 str[i++] = ch;
 str[i] = '\0';
 return i;
}
```

**D:** Perché strcmp restituisce un numero che è minore, uguale o maggiore di zero? Inoltre il valore restituito ha qualche significato? [p. 304]

**R:** Il valore restituito dalla strcmp probabilmente differisce dal quello della versione tradizionale della funzione. Considerate la versione presente nel libro *The C Programming Language* di Kernighan e Ritchie:

```
int strcmp(char *s, char *t)
{
 int i;

 for (i = 0; s[i] == t[i]; i++)
 if (s[i] == '\0')
 return 0;
 return s[i] - t[i];
}
```

Il valore restituito è la differenza tra i primi caratteri che differiscono nelle stringhe s e t. Questo valore sarà negativo se s punta a una stringa "minore" di quella puntata da t. Il valore sarà positivo se s punta a una stringa "maggiore". Tuttavia non vi sono garanzie che la strcmp sia effettivamente scritta in questo modo e quindi è meglio non assumere che la magnitudine del valore restituito abbia qualche particolare significato.

**D: Il nostro compilatore genera un messaggio di warning quando cerchiamo di compilare l'istruzione while presente nella funzione strcat:**

```
while (*p++ = *s2++)
;
```

**Cosa stiamo sbagliando?**

**R:** Nulla. Molti compilatori (ma non tutti) generano un messaggio di warning se viene usato = dove normalmente ci si aspetterebbe un ==. Questo messaggio è valido almeno nel 95% dei casi e ci risparmierà molte operazioni di debugging. Sfortunatamente l'avvertimento non è rilevante in questo particolare esempio. Infatti vogliamo effettivamente utilizzare l'operatore = e non l'operatore ==. Per sbarazzarci del messaggio riscriviamo il ciclo while in questo modo:

```
while ((*p++ = *s2++) != 0)
;
```

Dato che while solitamente controlla se \*p++ = \*s2++ è diverso da 0, non abbiamo modificato il significato dell'istruzione. Il messaggio viene evitato perché l'istruzione adesso controlla una condizione e non un assegnamento. Con il compilatore GCC, mettere un paio di parentesi attorno all'assegnamento è un altro modo per evitare il messaggio di warning:

```
while ((*p++ = *s2++) != 0)
;
```

**D: Le funzioni strlen e strcat sono effettivamente scritte come sono presentate nella Sezione 13.6?**

**R:** È possibile, sebbene tra i produttori di compilatori sia pratica comune scrivere queste funzioni (e molte altre funzioni sulle stringhe) in linguaggio assembly invece che in C. Le funzioni per le stringhe hanno bisogno di essere il più veloci possibile dato che spesso vengono utilizzate per gestire stringhe di lunghezza arbitraria. Scrivere queste funzioni in assembly permette di raggiungere una grande efficienza sfruttando le speciali istruzioni che le CPU possono fornire per la gestione delle stringhe.

**D: Perché lo standard C utilizza il termine "parametri di programma" invece che "argomenti della riga di comando"? [p. 314]**

**R:** I programmi non vengono sempre eseguiti dalla riga di comando. In una tipica interfaccia grafica, per esempio, i programmi vengono lanciati con un clic del mouse. In un ambiente di questo tipo non c'è una riga di comando tradizionale sebbene ci siano altri modi per passare informazioni al programma. Il termine "parametri di programma" lascia aperta la porta a tutte queste alternative.

**D: Dobbiamo utilizzare i nomi argc e argv per i parametri del main? [p. 314]**

**R:** No. L'utilizzo dei nomi argc e argv è solamente una convenzione, non un obbligo del linguaggio.

**D: Abbiamo visto argv dichiarato come \*\*argv invece di \*argv[]. È ammисibile?**

**R:** Certamente. Quando viene dichiarato un parametro, scrivere \*a è sempre equivalente a scrivere a[], indipendentemente dal tipo degli elementi di a.

**D:** Abbiamo visto come creare un vettore i cui elementi sono dei puntatori a stringhe letterali. Ci sono altre applicazioni per i vettori di puntatori?

**R:** Si. Sebbene ci siamo focalizzati sul vettore di puntatori a stringhe di caratteri, questa non è l'unica applicazione per i vettori di puntatori. Potremmo avere facilmente un vettore i cui elementi puntino ad altri tipi di dato. I vettori di puntatori sono particolarmente utili se utilizzati in congiunzione con l'allocazione dinamica della memoria [allocazione dinamica della memoria > 17.1].

## Esercizi

### Sezione 13.3

1. Ci si aspetta che le seguenti chiamate a funzione stampino un singolo carattere new-line, ma alcune non sono corrette. Identificate quali chiamate non funzionano e spiegate perché.

- |                                      |                                 |
|--------------------------------------|---------------------------------|
| (a) <code>printf("%c", '\n');</code> | (g) <code>putchar('\n');</code> |
| (b) <code>printf("%c", "\n");</code> | (h) <code>putchar("\n");</code> |
| (c) <code>printf("%s", '\n');</code> | (i) <code>puts('\n');</code>    |
| (d) <code>printf("%c", "\n");</code> | (j) <code>puts("\n");</code>    |
| (e) <code>printf('\n');</code>       | (k) <code>puts("");</code>      |
| (f) <code>printf("\n");</code>       |                                 |

- W 2. Supponete che la variabile `p` sia stata dichiarata in questo modo:

```
char *p = "abc";
```

Quale delle seguenti chiamate a funzione sono ammesse? Mostrate l'output prodotto da ognuna delle chiamate ammesse e spiegate perché le altre non lo sono.

- |                               |
|-------------------------------|
| (a) <code>putchar(p);</code>  |
| (b) <code>putchar(*p);</code> |
| (c) <code>puts(p);</code>     |
| (d) <code>puts(*p);</code>    |

3. \*Supponete di chiamare la funzione `scanf` in questo modo:

```
scanf("%d%s%d", &i, s, &j);
```

Se l'utente immette 12abc34 56def78, quali saranno i valori assunti da `i`, `s` e `j` dopo la chiamata? (Assumete che `i` e `j` siano variabili `int` e che `s` sia un vettore di caratteri).

- W 4. Modificate la funzione `read_line` in ognuno dei seguenti modi:

- Fate in modo che salti tutti gli spazi bianchi prima di iniziare a salvare i caratteri di input.
- Fate in modo che la lettura si interrompa al primo carattere di spazio bianco.  
*Suggerimento:* per determinare se il carattere è uno spazio bianco o meno chiamate la funzione `isspace` [funzione `isspace` > 23.5].
- Fate in modo che la lettura venga interrotta non appena si incontra un carattere new-line e che questo venga memorizzato nella stringa.
- I caratteri per i quali non c'è spazio a sufficienza per memorizzarli devono essere lasciati al loro posto.

**Sezione 13.4** 5. (a) Scrivete una funzione chiamata `capitalize` che trasforma in maiuscole tutte le lettere contenute nel suo argomento. L'argomento sarà costituito da una stringa terminante con il carattere null e contenente un numero arbitrario di caratteri (non solo lettere). Utilizzate l'indicizzazione dei vettori per accedere ai caratteri presenti nella stringa. Suggerimento: per convertire i caratteri utilizzate la funzione `toupper` [funzione `toupper` > 23.5].

(b) Riscrivete la funzione `capitalize` utilizzando l'aritmetica dei puntatori per accedere ai caratteri contenuti nella stringa.

W 6. Scrivete una funzione chiamata `censor` che modifichi una stringa rimpiazzando ogni occorrenza di `foo` con `xxx`. Per esempio: la stringa "food fool" dovrà diventare "xxd xxxl". Fate in modo che la funzione sia più corta possibile senza sacrificare la chiarezza.

**Sezione 13.5** 7. Supponete che `str` sia un vettore di caratteri. Quale delle seguenti istruzioni non è equivalente alle altre tre?

- (a) `*str = 0;`
- (b) `str[0] = '\0';`
- (c) `strcpy(str, "");`
- (d) `strcat(str, "");`

W 8. \*Quale sarà il valore della stringa `str` dopo l'esecuzione delle seguenti istruzioni?  
`strcpy(str, "tire-bouchon");`  
`strcpy(&str[4], "d-or-wi");`  
`strcat(str, "red?");`

9. Quale sarà il valore della stringa `s1` dopo l'esecuzione delle seguenti istruzioni?  
`strcpy(s1, "computer");`  
`strcpy(s2, "science");`  
`if (strcmp(s1, s2) < 0)`  
`strcat(s1, s2);`  
`else`  
`strcat(s2, s1);`  
`s1[strlen(s1)-6] = '\0';`

W 10. Ci si aspetta che la funzione seguente crei una copia identica di una stringa. Cosa c'è di sbagliato nella funzione?

```
char *duplicate(const char *p)
{
 char *q;
 strcpy(q, p);
 return q;
}
```

11. La Sezione D&R alla fine di questo capitolo mostra come la funzione `strcmp` possa essere scritta utilizzando l'indicizzazione dei vettori. Modificate la funzione in modo da utilizzare l'aritmetica dei puntatori.

**12.** Scrivete la seguente funzione:

```
void get_extension(const char *file_name, char *extension);
```

file\_name punta a una stringa contenente il nome di un file. La funzione dovrebbe salvare l'estensione del file nella stringa puntata da extension. Per esempio, se il nome del file è "memo.txt" allora la funzione dovrà salvare "txt" all'interno della stringa puntata da extension. Se il nome del file è sprovvisto di estensione, la funzione dovrà memorizzare una stringa vuota (un singolo carattere null) nella stringa puntata da extension. Mantenete la funzione il più semplice possibile utilizzando le funzioni strlen e strcpy.

**13.** Scrivete la funzione seguente:

```
void build_index_url(const char *domain, char *index_url);
```

domain punta a una stringa contenente un dominio internet come "knking.com". La funzione dovrà aggiungere "http://www." all'inizio della stringa e "/index.html" alla fine. Il risultato dovrà essere memorizzato nella stringa puntata da index\_url (con questo esempio il risultato sarà "http://www.knking.com/index.html"). Potete assumere che index\_url punti a una variabile che sia sufficientemente lunga da contenere la stringa risultante. Mantenete la funzione il più semplice possibile utilizzando le funzioni strcat e strcpy.

**14. \***Cosa stampa il seguente programma?

```
#include <stdio.h>

int main(void)
{
 char s[] = "Hsjodi", *p;
 for (p = s; *p; p++)
 --*p;
 puts(s);
 return 0;
}
```

**15. \***Sia f la seguente funzione:

```
int f(char *s, char *t)
{
 char *p1, *p2;
 for (p1 = s; *p1; p1++)
 for (p2 = t; *p2; p2++)
 if (*p1 == *p2) break;
 if (*p2 == '\0') break;
 return p1 - s;
}
```

(a) Qual è il valore di f("abcd", "babc"); ?

(b) Qual è il valore di f("abcd", "bcd"); ?

(c) In generale cosa restituisce f quando le vengono passate le due stringhe s e t?

- W 16. Utilizzate la tecnica della Sezione 13.6 per condensare la funzione `count_spaces` della Sezione 13.4. In particolare rimpiazzate l'istruzione `for` con un ciclo `while`.

17. Scrivete la seguente funzione:

```
bool test_extension(const char *file_name
const char *extension);
```

`file_name` punta a una stringa contenente il nome di un file. La funzione dovrà restituire `true` se l'estensione del file combacia con la stringa puntata da `extension` non facendo caso al fatto che le lettere siano maiuscole o minuscole. Per esempio: la chiamata `test_extension("memo.txt", "TXT")`; dovrà restituire `true`. Incorporate nella vostra funzione l'idioma per la "ricerca della fine di una stringa". Suggerimento: utilizzate la funzione `toupper` [funzione `toupper` > 23.5] per convertire i caratteri nella forma maiuscola prima di fare il confronto.

18. Scrivete la funzione

```
void remove_filename(char *url);
```

`url` punta a una stringa contenente una URL (*Uniform Resource Locator*) che termina con il nome di un file (come "`http://www.knking.com/index.html`"). La funzione dovrà modificare la stringa rimuovendo il nome del file e la barra (*slash*) che lo precede (nel nostro esempio il risultato sarebbe "`http://www.knking.com`"). Incorporate nella funzione l'idioma di "ricerca della fine di una stringa". Suggerimento: rimpiazzate l'ultima barra presente nella stringa con un carattere null.

## Progetti di programmazione

1. Scrivete un programma che cerchi la "maggiore" e la "minore" tra una serie di parole. Dopo che l'utente avrà immesso le parole, il programma dovrà determinare quali verranno prima e quali dopo secondo l'ordine alfabetico. Il programma dovrà smettere di accettare altro input nel momento in cui l'utente immette una parola di quattro lettere. Assumete che non ci siano parole con più di 20 lettere. Una sessione interattiva del programma potrebbe presentarsi in questo modo:

```
Enter word: dog
Enter word: zebra
Enter word: rabbit
Enter word: catfish
Enter word: walrus
Enter word: cat
Enter word: fish
Smallest word: cat
Largest word: zebra
```

Suggerimento: utilizzate due stringhe chiamate `smallest_word` e `largest_word` per tenere traccia della parola "maggiore" e di quella "minore" tra quelle immesse fino a quel momento. Ogni volta che l'utente immetterà una nuova parola utiliziate la `strcmp` per confrontarla con `smallest_word`. Se la nuova parola è "minore",

allora utilizzate la funzione `strcpy` per salvarla all'interno di `smallest_word`. Eseguite un confronto simile con `largest_word`. Utilizzate `strlen` per determinare quando l'utente ha immesso una parola di quattro lettere.

2. Migliorate il programma `remind.c` della Sezione 13.5 in questo modo:
  - (a) Fate in modo che il programma stampi un messaggio di errore e ignori un promemoria se il giorno corrispondente è negativo o maggiore di 31. *Suggerimento:* utilizzate l'istruzione `continue`.
  - (b) Fate in modo che l'utente possa immettere un giorno, un orario espresso in 24 ore e un promemoria. Stampate la lista dei promemoria ordinandoli per giorno e poi per ora (il programma originale permette che l'utente possa scrivere l'orario ma questo viene trattato come parte del promemoria).
  - (c) Fate in modo che il programma stampi la lista dei promemoria di *un anno*. Questo richiede che l'utente immetta i giorni nel formato *mese/giorno*.
3. Modificate il programma `deal.c` della Sezione 8.2 in modo che stampi i nomi completi delle carte che gestisce:

Enter number of cards in hand: 5

Your hand:

Seven of clubs

Two of spades

Five of diamonds

Ace of spades

Two of hearts

*Suggerimento:* rimpiazzate `rank_code` e `suit_code` con vettori contenenti dei puntatori a stringhe.

4. Scrivete un programma chiamato `reverse.c` che faccia l'echo degli argomenti della riga di comando ripresentandoli in ordine inverso. Eseguire il programma scrivendo  
`reverse void and null`  
 dovrà produrre il seguente output:  
`null and void`
5. Scrivete un programma chiamato `sum.c` che faccia la somma degli argomenti della riga di comando (si assume che siano interi). Eseguire il programma scrivendo  
`sum 8 24 62`  
 dovrà produrre il seguente risultato.  
`Total: 94`  
*Suggerimento:* utilizzate la funzione `atoi` [funzione `atoi` > 26.2] per convertire gli argomenti della riga di comando dal formato stringa al formato intero.
6. Migliorate il programma `planet.c` della Sezione 13.7 in modo che, durante il confronto degli argomenti della riga di comando con le stringhe presenti nel vettore `planets`, ignori il fatto che le lettere siano minuscole o maiuscole.

7. Modificate il Progetto di programmazione 11 del Capitolo 5 in modo che utilizzi dei vettori contenenti dei puntatori a delle stringhe invece di istruzioni switch. Per esempio, invece di utilizzare un'istruzione switch per stampare la parola corrispondente alla prima cifra, utilizzate la cifra come indice di un vettore contenente le stringhe "twenty", "thirty" e così via.
8. Modificate il Progetto di programmazione 5 del Capitolo 7 in modo da includere la seguente funzione:

```
int compute_scrabble_value(const char *word);
```

La funzione dovrà restituire il punteggio associato alla stringa puntata da word.

9. Modificate il Progetto di Programmazione 10 del Capitolo 7 in modo da includere la seguente funzione:

```
int compute_vowel_count(const char *sentence);
```

Il programma dovrà restituire il numero di vocali presenti nella stringa puntata dal parametro sentence.

10. Modificate il Progetto di programmazione 11 del Capitolo 7 in modo da includere la seguente funzione:

```
void reverse_name(char *name);
```

La funzione si aspetta che name punti a una stringa contenente un nome seguito da un cognome. La funzione modifica la stringa originale in modo che per primo venga presentato il cognome, seguito da una virgola, uno spazio, l'iniziale del nome e un punto. La stringa originale può contenere degli spazi aggiuntivi prima del nome, tra il nome e il cognome, e dopo il cognome.

11. Modificate il Progetto di programmazione 13 del Capitolo 7 in modo da includere la seguente funzione:

```
double compute_average_word_length(const char *sentence);
```

La funzione restituisce la lunghezza media delle parole contenute nella stringa puntata da sentence.

12. Modificate il Progetto di programmazione 14 del Capitolo 8 in modo che durante la lettura della frase salvi le parole in un vettore bidimensionale di char. Ogni riga del vettore dovrà contenere una singola parola. Assumete che la frase non contenga più di 30 parole e che non ci siano parole più lunghe di 20 caratteri. Assicuratevi di memorizzare il carattere null alla fine di ogni parola in modo da poterla trattare come una stringa.

13. Modificate il Progetto di programmazione 15 del Capitolo 8 in modo che includa la seguente funzione:

```
void encrypt(char *message, int shift);
```

La funzione si aspetta che message punti a una stringa contenente un messaggio cifrato. Il parametro shift rappresenta lo sfasamento che deve essere applicato alle lettere del messaggio.

14. Modificate il Progetto di programmazione 16 del Capitolo 8 in modo che includa la seguente funzione:

```
bool are_anagrams(const char *word1, const char *word2);
```

La funzione restituisce true se la stringa puntata da word1 e quella puntata da word2 sono anagrammi.

15. Modificate il Progetto di programmazione 6 del Capitolo 10 in modo che includa la seguente funzione:

```
int evaluate_RPN_expression(const char *expression);
```

La funzione restituisce il valore dell'espressione RPN puntata dal parametro expression.

16. Modificate il Progetto di programmazione 1 del Capitolo 12 in modo che includa la seguente funzione:

```
void reverse(char *message);
```

La funzione inverte la stringa puntata da message. Suggerimento: utilizzate due puntatori, uno che punta inizialmente al primo carattere della stringa e l'altro che inizialmente punta all'ultimo carattere. Fate in modo che la funzione inverta questi caratteri e sposti i puntatori l'uno verso l'altro, ripetendo il processo fino a quando questi non si incontrano.

17. Modificate il Progetto di Programmazione 2 del Capitolo 12 in modo che includa la seguente funzione:

```
bool is_palindrome(const char *message);
```

La funzione restituisce true se la stringa puntata dal parametro message è palindroma.

18. Scrivete un programma che accetti una data dall'utente nel formato mm/gg/aaaa e poi la stampi nel formato mese gg, aaaa:

Enter a date (mm/dd/yyyy): 2/17/2011

You entered the date February 17, 2011

Memorizzate i nomi dei mesi in un vettore contenente puntatori a stringhe.

# 14 Il preprocessore

Nei precedenti capitoli abbiamo usato le direttive `#define` e `#include` senza entrare nel dettaglio di quello che fanno. Queste direttive (e altre che non abbiamo ancora trattato) sono gestite dal **preprocessore**, un software che manipola i programmi C immediatamente prima della compilazione. L'affidarsi a un preprocessore rende il C (assieme al C++) unico tra i maggiori linguaggi di programmazione.

Il preprocessore è uno strumento molto potente, tuttavia può essere la causa di bachi difficili da individuare. Inoltre può essere facilmente utilizzato male creando programmi quasi impossibili da comprendere. Nonostante ciò alcuni programmati C si affidano pesantemente al preprocessore, anche se è preferibile farne ricorso con moderazione.

Questo capitolo inizia con una descrizione di come opera il preprocessore (Sezione 14.1) e poi dà alcune regole generali che influenzano tutte le direttive di preprocessamento (Sezione 14.2). Le Sezioni 14.3 e 14.4 trattano due delle più importanti capacità del preprocessore: la definizione delle macro e la compilazione condizionale (rimandiamo al Capitolo 15 la trattazione dettagliata dell'inclusione dei file e della altre capacità più importanti). La Sezione 14.5 discute le direttive meno utilizzate del preprocessore: `#error`, `#line` e `#pragma`.

## 14.1 Come opera il preprocessore

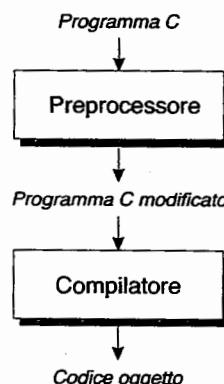
Il comportamento del preprocessore è controllato dalle **direttive di preprocessamento**: dei comandi che iniziano con il carattere `#`. Nei precedenti capitoli abbiamo incontrato due di queste direttive: `#define` e `#include`.

La direttiva `#define` definisce una **macro** (un nome che rappresenta qualcos'altro, come una costante o un'espressione utilizzata di frequente). Il preprocessore risponde alle direttive `#define` memorizzando il nome della macro assieme alla sua definizione. Quando in un secondo momento la macro viene utilizzata, il preprocessore la "espande" rimpiazzandola con il suo valore.

La direttiva `#include` dice al preprocessore di aprire un particolare file e di "includere" il suo contenuto come parte del file che deve essere compilato. Per esempio, la linea  
`#include <stdio.h>`

indica al preprocessore di aprire il file chiamato stdio.h e di immettere il suo contenuto all'interno del programma (tra le altre cose, stdio.h contiene i prototipi per le funzioni standard di input/output del C).

Il diagramma seguente illustra il ruolo del preprocessore nel processo di compilazione:



L'input del preprocessore è un programma C che può contenere delle direttive. Durante il processo il preprocessore esegue queste direttive rimuovendole. L'output del preprocessore è un altro programma C: una versione modificata del programma originale priva di direttive. L'output va direttamente "in pasto" al compilatore, il quale controlla il programma alla ricerca di eventuali errori e lo traduce in codice oggetto (istruzioni macchina).

Per vedere cosa fa il preprocessore, usiamolo sul programma celsius.c della Sezione 2.6. Ecco il programma originale:

```

/*
 * Converte una temperatura in gradi Fahrenheit in una temperatura in gradi Celsius
 */
#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void)
{
 float fahrenheit, celsius;

 printf("Enter Fahrenheit temperature: ");
 scanf("%f", &fahrenheit);

 celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
 printf("Celsius equivalent: %.1f\n", celsius);

 return 0;
}

```

Dopo il preprocessamento, il programma si presenterà in questo modo:

```
Riga vuota
Riga vuota
Righe prese da stdio.h
Riga vuota
Riga vuota
Riga vuota
Riga vuota
int main(void)
{
 float fahrenheit, celsius;
 printf("Enter Fahrenheit temperature: ");
 scanf("%f", &fahrenheit);
 celsius = (fahrenheit - 32.0f) * (5.0f / 9.0f);
 printf("Celsius equivalent: %.1f\n", celsius);
 return 0;
}
```

Il preprocessore ha risposto alla direttiva `#include` aggiungendo il contenuto di `stdio.h`. Il preprocessore ha rimosso inoltre le direttive `#define` e ha rimpiazzato `FREEZING_PT` e `SCALE_FACTOR` ogni volta che compaiono all'interno del file. Osservate che il preprocessore non rimuove le righe contenenti le direttive ma semplicemente le svuota.

Come illustra questo esempio, il preprocessore non solo esegue le direttive, ma in particolare sostituisce ogni commento con un singolo carattere di spazio. Alcuni preprocessori vanno oltre rimuovendo gli spazi bianchi non necessari e le tabulazioni all'inizio delle righe indentate.

Agli albori del C, il preprocessore era un programma separato che alimentava il compilatore con il suo output; oggi è spesso parte del compilatore e alcune porzioni del suo output potrebbero non essere necessariamente del codice C (per esempio includere un header standard come `<stdio.h>` può rendere le sue funzioni disponibili senza necessariamente copiare il contenuto dell'header all'interno del codice del programma). Nonostante ciò è utile pensare al preprocessore come un componente separato rispetto al compilatore. Infatti la maggior parte dei compilatori C forniscono un modo per visualizzare l'output prodotto dal preprocessore. Certi compilatori generano l'output del preprocessore quando vengono specificate alcune opzioni (GCC si comporta in questo modo quando viene usata l'opzione `-E`). Altri compilatori, invece, sono provvisti di un programma separato che si comporta come il preprocessore integrato. Controllate la documentazione del vostro compilatore per maggiori informazioni.

Attenzione: il preprocessore possiede solo una conoscenza limitata del C, per questo quando esegue le direttive è in grado di creare programmi non ammissibili. Spesso il programma originale sembra corretto, cosa che rende questi errori ancora più difficili da trovare. Nei programmi complicati esaminare l'output del preprocessore può rivelarsi utile per localizzare questo tipo di errori.

## 14.2 Direttive del preprocessore

La maggior parte delle direttive del preprocessore ricade in una delle seguenti categorie:

- **Definizione di macro.** La direttiva `#define` definisce una macro mentre la direttiva `#undef` rimuove la definizione di una macro.
- **Inclusione di file.** La direttiva `#include` fa sì che il contenuto di un file specificato sia incluso all'interno del programma.
- **Compilazione condizionale.** Le directive `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` e `#endif` permettono che alcune porzioni di testo vengano incluse o escluse da un programma a seconda delle condizioni che possono essere analizzate dal preprocessore.

Le direttive rimanenti (`#error`, `#line` e `#pragma`) sono maggiormente specializzate e per questo vengono utilizzate più raramente. Dedicheremo il resto di questo capitolo a un esame approfondito delle direttive del preprocessore. L'unica che non discuteremo in dettaglio è la direttiva `#include` che invece è trattata nella Sezione 15.2.

Prima di proseguire vediamo alcune regole che si applicano a tutte le direttive:

- **Le direttive iniziano sempre con il simbolo #.** Il simbolo `#` non deve necessariamente trovarsi all'inizio della riga ma può essere preceduto da spazio bianco. Dopo tale simbolo si trova il nome della direttiva seguito da tutte le informazioni di cui quest'ultima può avere bisogno.
- **I token di una direttiva possono essere separati da un numero qualsiasi di spazi e tabulazioni orizzontali.** La direttiva seguente, per esempio, è ammessa:

```
define N 100
```

- **Le direttive terminano sempre al primo carattere new-line a meno che non sia specificato altrimenti.** Per continuare una direttiva nella riga seguente dobbiamo terminare la riga corrente con il carattere `\`. La seguente direttiva, per esempio, definisce una macro che rappresenta la capacità di un hard disk misurata in byte:

```
#define DISK_CAPACITY (SIDES * \
 TRACKS_PER_SIDE * \
 SECTORS_PER_TRACK * \
 BYTES_PER_SECTOR)
```

- **Le direttive possono trovarsi in qualsiasi punto di un programma.** Sebbene solitamente le direttive `#define` e `#include` vengano messe all'inizio di un file, per le altre direttive è molto più probabile comparire successivamente, anche nel mezzo della definizione di una funzione.
- **I commenti possono trovarsi nella stessa riga di una direttiva.** Infatti è una buona pratica mettere un commento alla fine della definizione di una macro per spiegare il suo significato:

```
#define FREEZING_PT 32.0F /* punto di congelamento dell'acqua */
```

## 14.3 Definizione di macro

Le macro che stiamo utilizzando fin dal Capitolo 2 sono conosciute come *macro semplici* per il fatto che sono prive di parametri. Il preprocessore supporta anche le *macro parametriche*. Tratteremo prima le macro semplici e poi quelle parametriche. Dopo averle trattate separatamente esamineremo le proprietà condivise da entrambe le categorie.

### Macro semplici

La definizione di una **macro semplice** (lo standard C le chiama **object-like macro**) ha la forma:

```
#define identificatore elenco-di-sostituzione
```

L'elenco di sostituzione è una qualsiasi sequenza di **token del preprocessore** che sono simili ai token discussi nella Sezione 2.8. Ogni volta che in questo capitolo utilizzeremo la parola **token** intenderemo un "token per il preprocessore".

L'elenco di sostituzione di una macro può contenere identificatori, keyword, costanti numeriche, costanti carattere, stringhe letterali, operatori e caratteri di interpuzione. Quando incontriamo la definizione di una macro, il preprocessore prende nota del fatto che *l'identificatore* rappresenta *l'elenco di sostituzione*. Ogni volta che nella parte successiva del file viene incontrato *l'identificatore*, questo viene sostituito con *l'elenco di sostituzione*.



Non mettete simboli aggiuntivi all'interno della definizione di una macro, questi diventerebbero parte della lista di sostituzione. Mettere il simbolo = nella definizione di una macro è un errore piuttosto comune:

```
#define N = 100 /** SBAGLIATO **/
```

---

```
int a[N]; /* diventa int a[= 100]; */
```

In questo esempio abbiamo erroneamente definito N come una coppia di token (= e 100). Un altro errore frequente è quello di terminare la definizione di una macro con un punto e virgola:

```
#define N 100; /** SBAGLIATO **/
```

---

```
int a[N]; /* diventa int a[100;; */
```

Con questa definizione N corrisponde ai token 100 e ;.

Il compilatore si accorgerà della maggior parte degli errori causati da simboli aggiunti nelle definizioni delle macro. Sfortunatamente il compilatore segnalerà come un errore ogni utilizzo della macro invece di segnalare il vero colpevole (la definizione della macro) che è stato rimosso dal preprocessore.

Le macro semplici sono utilizzate principalmente per definire quello che Kernighan e Ritchie chiamavano "costanti manifeste". Utilizzando le macro possiamo assegnare nomi ai valori numerici, a caratteri e a stringhe.

```
#define STR_LEN 80
#define TRUE 1
#define FALSE 0
#define PI 3.14159
#define CR '\r'
#define EOS '\0'
#define MEM_ERR "Error: not enough memory"
```

Utilizzare `#define` per assegnare dei nomi alle costanti ha diversi vantaggi significativi.

- **Rende i programmi più facili da leggere.** Il nome della macro (se scelto bene) aiuta il lettore a comprendere il significato della costante. L'alternativa è un programma pieno di "numeri magici" che possono disorientare facilmente il lettore.
- **Rende i programmi più facili da modificare.** Modificando la sola definizione della macro possiamo cambiare il valore di una costante in tutto il programma. Le costanti codificate "in modo fisso" sono molto più difficili da modificare, soprattutto se qualche volta compaiono in una forma leggermente modificata (per esempio, un programma con un vettore di lunghezza 100 può avere un ciclo che va da 0 a 99. Se cercassimo semplicemente le occorrenze di 100 all'interno del programma, non troveremmo il 99).
- **Aiuta a evitare inconsistenze ed errori tipografici.** Se una costante numerica come 3.14159 compare diverse volte, ci sono buone probabilità che venga scritta per errore come 3.1416 o 3.14195.

Sebbene le macro semplici vengano utilizzate molto spesso per definire il nome delle costanti, possono essere utilizzate anche per altri scopi.

- **Effettuare dei piccoli cambiamenti alla sintassi del C.** In effetti possiamo alterare la sintassi del C definendo delle macro da utilizzare come nomi alternativi per i simboli del C. Per esempio, i programmati che preferiscono i token `begin` ed `end` del Pascal alle parentesi { e } del C possono definire le seguenti macro:

```
#define BEGIN {
#define END }
```

- **Rinominare i tipi.** Nella Sezione 5.2 abbiamo creato un tipo booleano rinominando il tipo `int`:

```
#define BOOL int
```

Sebbene certi programmati utilizzino le macro per questo scopo, le definizioni di tipo [definizioni di tipo > 7.5] sono un metodo migliore.

- **Controllare la compilazione condizionale.** Come vedremo nella Sezione 14.4, le macro giocano un ruolo importante nella compilazione condizionale. Per esempio, la presenza della seguente riga in un programma può indicare che questo debba essere compilato nella "modalità di debug", ovvero con istruzioni aggiuntive per produrre dell'output utile per il debugging:

```
#define DEBUG
```

Per inciso possiamo dire che è possibile avere delle macro con l'elenco di sostituzione vuoto, così come vediamo nell'esempio appena presentato.

Di solito i programmati C hanno l'abitudine di utilizzare solo lettere maiuscole per i nomi delle macro che vengono utilizzate come costanti. Tuttavia non vi è consenso su come scrivere le macro utilizzate per altri scopi. Dato che queste (specialmente quelle parametriche) sono fonte di bachi, ad alcuni programmati piace attirare l'attenzione su di esse utilizzando solo lettere maiuscole per i loro nomi. Altri programmati preferiscono seguire lo stile del libro *The C Programming Language* di Kernighan e Ritchie che per le macro utilizza dei nomi costituiti da lettere minuscole.

## Macro parametriche

La definizione di una **macro parametrica** (conosciuta anche come **function-like macro**) ha la forma

```
#define identificatore(x1, x2, ..., xn) elenco-di-sostituzione
```

dove  $x_1, x_2, \dots, x_n$  sono degli identificatori (**parametri** della macro). I parametri possono comparire nell'elenco di sostituzione quante volte si desidera.



*Non devono esserci spazi tra il nome della macro e la parentesi tonda sinistra. Se viene lasciato dello spazio, il preprocessore penserà che si stia definendo una macro semplice e tratterà  $(x_1, x_2, \dots, x_n)$  come parte dell'elenco di sostituzione.*

Quando il preprocessore incontra la definizione di una macro parametrica, memorizza la sua definizione per gli utilizzi successivi. Ogni volta che un'**invocazione** della macro della forma *identificatore*( $y_1, y_2, \dots, y_n$ ) compare nel programma (dove  $y_1, y_2, \dots, y_n$  sono sequenze di token), il preprocessore la rimpiazza con l'elenco di sostituzione rimpiazzando  $x_1$  con  $y_1$ ,  $x_2$  con  $y_2$ , e così via.

Per esempio, supponete di aver definito le seguenti macro:

```
#define MAX(x,y) ((x)>(y)?(x):(y))
#define IS_EVEN(n) ((n)%2==0)
```

(Il numero di parentesi presenti in queste macro può sembrare eccessivo, tuttavia, come vedremo più avanti in questa sezione, vi è una ragione ben precisa.) Supponete ora di invocare le due macro in questo modo:

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

Il preprocessore rimpiazzerà queste righe con

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

Come mostra questo esempio, spesso le macro parametriche fungono da semplici funzioni. MAX si comporta come una funzione che calcola il maggiore tra due numeri.

Il **WIN** si comporta come una funzione che restituisce 1 se il suo argomento è un numero pari, altrimenti restituisce 0.

Ecco una macro più complessa che si comporta come una funzione:

```
#define TOUPPER(c) ('a'<=(c)&&(c)<='z'?((c)-'a'+'A':(c))
```

Questa macro controlla se il carattere c è compreso tra 'a' e 'z'. Se è così produce la versione maiuscola di c sottraendo 'a' e sommando 'A'. Se non è così, la macro non modifica c (l'header `<ctype.h>` [header <ctype.h> 23.5] fornisce una funzione simile chiamata `toupper` che è più portabile).

Una macro parametrica può avere un elenco di parametri vuoto. Ecco un esempio:

```
#define getch() getc(stdin)
```

L'elenco vuoto di parametri non è necessario ma fa in modo che `getchar` somigli a una funzione (sì, è la stessa `getchar` che appartiene a `<stdio.h>`). Vedremo nella Sezione 22.4 che di solito la `getchar` è implementata come una macro oltre che come una funzione). Utilizzare una macro parametrica in luogo di una vera funzione presenta due vantaggi

- **Il programma può essere leggermente più veloce.** Solitamente una chiamata a funzione è causa di *overhead* durante l'esecuzione del programma (le informazioni sul contesto devono essere salvate, gli argomenti devono essere copiati e così via). L'invocazione di una macro, d'altro canto, non richiede alcun overhead (osservate però che le funzioni inline [funzioni inline > 18.6] del C99 forniscono un modo per evitare questo overhead senza utilizzare le macro).
- **Le macro sono "generiche".** I parametri delle macro, a differenza dei parametri delle funzioni, non possiedono un tipo particolare. Come risultato una macro può accettare argomenti di qualsiasi tipo, a patto che il programma risultante dopo il preprocessamento sia valido. Per esempio, possiamo utilizzare la macro `MAX` per trovare il maggiore tra due valori di tipo `int`, `long`, `float`, `double` e così via.

Tuttavia le macro parametriche possiedono anche alcuni svantaggi.

- **Il codice compilato spesso è di maggiori dimensioni.** Ogni invocazione di macro provoca l'inserimento dell'elenco di sostituzione e quindi l'incremento delle dimensioni del sorgente del programma (e quindi del codice compilato). Più spesso viene utilizzata una macro e più l'effetto è pronunciato. Il problema si aggrava quando le invocazioni delle macro vengono annidate. Considerate quello che succede quando utilizziamo `MAX` per trovare il maggiore tra tre numeri:

```
n = MAX(i, MAX(j, k));
```

Ecco come si presenta l'istruzione dopo il preprocessamento:

```
n = (((i)>(((j)>(k)?(j):(k)))?(i):(((j)>(k)?(j):(k))));
```

- **Non viene controllato il tipo degli argomenti.** Quando una funzione viene invocata, il compilatore controlla ogni argomento per vedere se è del tipo appropriato. Nel caso non lo fosse, o l'argomento viene convertito nel tipo appropriato oppure il compilatore produce un messaggio di errore. Gli argomenti delle macro non vengono controllati dal preprocessore e quindi non vengono convertiti.

- **Non è possibile avere un puntatore a una macro.** Come vedremo nella Sezione 17.7, il C permette puntatori a funzione, un concetto che è piuttosto utile in alcune situazioni di programmazione. Le macro vengono rimosse durante il preprocessamento e quindi non c'è un concetto corrispondente di "puntatore a una macro". Il risultato è che le macro non possono essere utilizzate in queste situazioni.
- **Una macro può calcolare i suoi argomenti più volte.** Una funzione calcola i suoi argomenti solamente una volta. Una macro può calcolare i suoi argomenti due o più volte. Calcolare un argomento più di una volta può provocare un comportamento inaspettato se l'argomento possiede dei side effect. Considerate cosa succede se uno degli argomenti di MAX possiede un side effect:

```
n = MAX(i++, j);
```

Ecco come si presenta la stessa riga dopo il preprocessamento:

```
n = ((i++)>(j)?(i++):(j));
```

Se i è maggiore di j allora i verrà (erroneamente) incrementata due volte e a n verrà assegnato un valore non atteso.



Gli errori causati quando un argomento di una macro viene calcolato più di una volta possono essere difficili da trovare perché l'invocazione della macro sembra uguale a una chiamata a funzione. A peggiorare le cose c'è il fatto che una macro può funzionare correttamente la maggior parte delle volte, creando problemi solo con certi argomenti che possiedono dei side effect. Per prevenire inconvenienti è meglio evitare side effect negli argomenti.

Le macro parametriche non sono apprezzabili solo per la semplice simulazione delle funzioni. In particolare, vengono utilizzate spesso come pattern di segmenti di codice che noi stessi possiamo trovare ripetitivi. Supponete di annoiarvi nello scrivere

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

ogni volta che abbiamo bisogno di stampare un intero. Possiamo definire la seguente macro che rende più facile la visualizzazione degli interi:

```
#define PRINT_INT(n) printf("%d\n", n)
```

Una volta che PRINT\_INT è stata definita, il preprocessore convertirà la riga

```
PRINT_INT(i/j);
```

in

```
printf("%d\n", i/j);
```

## L'operatore #

Le definizioni delle macro possono contenere due speciali operatori: # e ##. Nessuno di questi viene riconosciuto dal compilatore, bensì vengono eseguiti durante il preprocessamento.

D&R

L'operatore # converte gli argomenti di una macro in una stringa letterale. Questo operatore può trovarsi solo nell'elenco di sostituzione di una macro parametrica (le operazioni eseguite dall'operatore # sono conosciute come *stringization*, un termine che di sicuro non troverete nel dizionario).

Ci sono diversi utilizzi dell'operatore #, ma noi ne considereremo solo uno. Supponete di aver deciso di utilizzare la macro PRINT\_INT come un modo conveniente per stampare i valori di variabili ed espressioni di tipo intero durante il debugging. L'operatore # dà la possibilità alla PRINT\_INT di etichettare ogni valore che stampa. Ecco la nostra nuova versione di PRINT\_INT:

```
#define PRINT_INT(n) printf(#n " = %d\n", n)
```

L'operatore # posto davanti a n indica al preprocessore di creare una stringa letterale a partire dagli argomenti di PRINT\_INT. Quindi l'invocazione

```
PRINT_INT(i/j);
```

diventerà

```
printf("i/j" " = %d\n", i/j);
```

Nella Sezione 13.1 abbiamo visto che il compilatore unisce automaticamente le stringhe letterali adiacenti, di conseguenza questa istruzione è equivalente a

```
printf("i/j = %d\n", i/j);
```

Quando un programma viene eseguito, la printf visualizza sia l'espressione i/j che il suo valore. Se per esempio i è uguale a 11 e j è uguale a 2, l'output sarà

```
i/j = 5
```

## L'operatore ##

L'operatore ## "incolla" assieme due token (degli identificatori, per esempio) in modo da formarne uno solo (infatti l'operatore ## è conosciuto come *token-pasting* ovvero come "incolla token"). Se uno degli operandi è il parametro di una macro, l'unione avviene dopo che il parametro è stato rimpiazzato dall'argomento corrispondente. Considerate la macro seguente:

```
#define MK_ID(n) i##n
```

Quando la macro MK\_ID viene invocata (per esempio con MK\_ID(1)), il preprocessore per prima cosa sostituisce il parametro n con l'argomento (1 nel nostro caso). Successivamente il preprocessore unisce i e 1 in modo da formare un singolo token (i1). La seguente dichiarazione utilizza MK\_ID per creare tre identificatori:

```
int MK_ID(1), MK_ID(2), MK_ID(3);
```

Dopo la fase di preprocessamento la dichiarazione diventa

```
int i1, i2, i3;
```

L'operatore `##` non è una delle caratteristiche più usate del preprocessore, infatti è difficile pensare a situazioni in cui sia necessario. Per cercare un impiego realistico per `##`, riconsideriamo la macro `MAX` che è stata descritta precedentemente all'interno di questa sezione. Questa macro non si comporta correttamente nel caso in cui i suoi argomenti abbiano dei side effect. L'alternativa all'utilizzo della macro `MAX` è la scrittura di una funzione `max`. Sfortunatamente di solito una sola funzione `max` non è sufficiente, potremmo aver bisogno di una funzione `max` con argomenti di tipo `int`, una con argomenti di tipo `float` e così via. Tutte queste versioni di `max` sarebbero identiche eccetto per il tipo degli argomenti e il tipo restituito e così può sembrare inutilmente faticoso definirne così tante.

La soluzione è quella di scrivere una macro che si espanda nella definizione di una funzione `max`. La macro avrà un solo argomento, `type`, che rappresenterà il tipo dell'argomento e del valore restituito. C'è solo un inconveniente: se utilizziamo la macro per creare più di una funzione `max`, il programma non verrà compilato (il C non permette che due funzioni abbiano lo stesso nome, se sono definite all'interno dello stesso file). Per risolvere questo problema utilizzeremo l'operatore `##` per creare un nome diverso per ogni versione di `max`. Ecco come si presenterà la macro:

```
define GENERIC_MAX(type) \
type type##_max(type x, type y) \
{ \
 return x > y ? x : y; \
}
```

Fate caso a come `type` e `_max` vengono uniti per formare il nome della funzione.

Supponete che vi capiti di aver bisogno di una funzione `max` che lavori con valori `float`. Ecco come potremmo usare la macro `GENERIC_MAX` per definire la funzione:

```
GENERIC_MAX(float);
```

Il preprocessore espanderà questa riga nel codice seguente:

```
float float_max(float x, float y) { return x > y ? x : y; }
```

## Proprietà generali delle macro

Dopo aver discusso sia delle macro semplici sia di quelle parametriche, possiamo trattare le regole che si applicano a entrambe.

- **L'elenco di sostituzione di una macro può contenere delle invocazioni ad altre macro.** Per esempio possiamo definire la macro `TWO_PI` in termini della macro `PI`:

```
#define PI 3.14159
#define TWO_PI (2*PI)
```

Quando il preprocessore incontra `TWO_PI` all'interno del programma, lo sostituisce con `(2*PI)`. Il preprocessore **scandisce** nuovamente l'elenco di sostituzione per

vedere se questo contiene delle invocazioni ad altre macro (PI in questo caso). Il preprocessore scansionerà l'elenco di sostituzione tutte le volte che è necessario per eliminare tutti i nomi delle macro.

- **Il preprocessore sostituisce solo token interi e non porzioni di questi.** Come risultato si ha che il preprocessore ignora i nomi di macro che sono inseriti all'interno di identificatori, costanti carattere e stringhe letterali. Per esempio, supponete che un programma contenga le righe seguenti:

```
#define SIZE 256
int BUFFER_SIZE
if (BUFFER_SIZE > SIZE)
puts("Error: SIZE exceeded");
```

dopo il preprocessamento, le righe si presenteranno in questo modo:

```
int BUFFER_SIZE
if (BUFFER_SIZE > 256)
puts("Error: SIZE exceeded");
```

L'identificatore BUFFER\_SIZE e la stringa "Error: SIZE exceeded" non vengono interessate dal preprocessamento, anche se entrambe contengono la parola SIZE.

- **Normalmente le definizioni delle macro rimangono valide fino alla fine del file nel quale compaiono.** Dato che le macro sono gestite dal preprocessore non obbediscono alle normali regole di scope. Una macro definita all'interno del corpo di una funzione non è locale a quella funzione, ma rimane definita fino alla fine del file.
- **Una macro non può essere definita due volte a meno che la nuova definizione non sia identica a quella vecchia.** Delle differenze negli spazi sono ammesse ma i token presenti nell'elenco di sostituzione (e i parametri se, ce ne fossero) devono essere gli stessi.
- **La definizione delle macro può essere rimossa con la direttiva #undef.** La direttiva #undef ha la forma

#undef identificatore

dove *identificatore* è il nome di una macro. Per esempio, la direttiva

```
#undef N
```

rimuove la definizione corrente della macro N (se N non è stata definita come una macro, la direttiva #undef non ha alcun effetto). Uno degli utilizzi di #undef è quello di rimuovere la definizione esistente di una macro in modo che le possa essere associate una nuova.

## Parentesi nelle definizioni delle macro

L'elenco di sostituzione presente nella definizione delle nostre macro era pieno di parentesi. È veramente necessario averne così tante? La risposta è un deciso sì. Se

usassimo meno parentesi, a volte potremmo ottenere dei risultati inattesi (e indesiderati).

Vi sono due regole da seguire quando si decide dove inserire le parentesi nella definizione di una macro. Per prima cosa, se l'elenco di sostituzione contiene un operatore deve essere sempre racchiuso tra parentesi tonde:

```
#define TWO_PI (2*3.14159)
```

Come seconda regola si ha che se la macro possiede dei parametri, questi devono essere posti tra parentesi ogni volta che compaiono nell'elenco di sostituzione:

```
#define SCALE(x) ((x)*10)
```

Senza le parentesi non possiamo garantire che il compilatore tratti l'elenco di sostituzione e gli argomenti come un'unica espressione. Il compilatore potrebbe applicare le regole di precedenza tra gli operatori e quelle dell'associatività in modi non prevedibili.

Per illustrare l'importanza delle parentesi attorno all'elenco di sostituzione di una macro, considerate la seguente definizione senza parentesi:

```
#define TWO_PI 2*3.14159
```

Durante il preprocessamento, l'istruzione

```
conversion_factor = 360/TWO_PI;
```

si trasforma in

```
conversion_factor = 360/2*3.14159;
```

La divisione verrà eseguita prima della moltiplicazione portando a un risultato non previsto.

Racchiudere tra parentesi l'elenco di sostituzione non è sufficiente, se la macro possiede dei parametri (ogni occorrenza di un parametro necessita allo stesso modo delle parentesi). Supponiamo, per esempio, che la macro SCALE sia definita in questo modo:

```
#define SCALE(x) (x*10) /* sono necessarie delle parentesi attorno a x */
```

Durante il preprocessamento, l'istruzione

```
j = SCALE(i+1);
```

diventa uguale a

```
j = (i+1*10);
```

Dato che la moltiplicazione ha precedenza rispetto all'addizione, questa istruzione è equivalente a

```
j = i+10;
```

Naturalmente quello che volevamo era

```
j = (i+1)*10;
```



La mancanza di parentesi nella definizione di una macro può causare alcuni degli errori più frustranti del C. Il programma solitamente compilerà e la macro sembrerà funzionare, fallendo solo nei punti meno opportuni.

## Creare macro più complesse

L'operatore virgola può essere utile per creare delle macro più sofisticate perché ci permette di creare l'elenco di sostituzione costituito da una serie di espressioni. Per esempio, la macro seguente legge una stringa e poi la stampa:

```
#define ECHO(s) (gets(s), puts(s))
```

Le chiamate alla gets e alla puts sono espressioni e quindi è assolutamente ammисibile combinarle con l'operatore virgola. Possiamo invocare ECHO come se fosse una funzione:

```
ECHO(str); /* diventa (gets(str), puts(str)); */
```

Invece di utilizzare l'operatore virgola avremmo potuto racchiudere le chiamate alla gets e alla puts all'interno di parentesi graffe per formare un'istruzione composta:

```
#define ECHO(s) { gets(s); puts(s); }
```

Sfortunatamente questo metodo non funziona altrettanto bene. Supponete di utilizzare ECHO in un'istruzione if:

```
if (echo_flag)
 ECHO(str);
else
 gets(str);
```

Sostituendo ECHO otteniamo il seguente risultato:

```
if (echo_flag)
 { gets(str); puts(str); };
else
 gets(str);
```

Il compilatore tratterà le prime due righe come un'istruzione if completa:

```
if (echo_flag)
 { gets(str); puts(str); }
```

Il punto e virgola seguente verrà trattato dal compilatore come un'istruzione vuota e verrà prodotto un messaggio di errore a causa della clausola else dato che questa non appartiene ad alcun if. Possiamo risolvere questo problema ricordandoci di non mettere un punto e virgola dopo le invocazioni a ECHO, ma a quel punto il programma comparirà strano.

L'operatore virgola risolve questo problema per la macro ECHO, ma non per tutte le macro. Supponete che una macro abbia bisogno di contenere una serie di *istruzioni* e non semplicemente una serie di *espressioni*. L'operatore virgola non è di aiuto. Può incollare espressioni ma non istruzioni. La soluzione è quella di circondare le istru-

zioni in un ciclo do che abbia la condizione falsa (e che quindi verrà eseguito una volta sola):

```
do { ... } while (0)
```

Osservate che l'istruzione do non è completa (necessita del punto e virgola alla fine). Per vedere questa tecnica in azione, la incorporiamo nella nostra macro ECHO:

```
#define ECHO(s) \
 do { \
 gets(s); \
 puts(s); \
 } while (0)
```

Quando ECHO viene usata, deve essere fatta seguire da un punto e virgola in modo da completare l'istruzione do:

```
ECHO(str); /* diventa do {gets(str); puts(str); } while(0); */
```

## Macro predefinite

Il C possiede diverse macro predefinite. Ogni macro rappresenta una costante intera o una stringa letterale. Come illustra la Tabella 14.1, queste macro forniscono delle informazioni riguardo la compilazione corrente o lo stesso compilatore.

**Tabella 14.1** Macro predefinite

| Nome                | Descrizione                                                |
|---------------------|------------------------------------------------------------|
| <code>_LINE_</code> | Numero della linea attualmente in compilazione             |
| <code>_FILE_</code> | Nome del file attualmente in compilazione                  |
| <code>_DATE_</code> | Data di compilazione (nel formato "Mmm dd yyyy")           |
| <code>_TIME_</code> | Ora di compilazione (nel formato "hh:mm:ss")               |
| <code>_STDC_</code> | 1 se il compilatore è conforme allo standard C (C89 o C99) |

Le macro `_DATE_` e `_TIME_` identificano l'istante di compilazione di un programma. Per esempio, supponete che un programma inizi con le seguenti istruzioni:

```
printf("Wacky Windows (c) 2010 Wacky Software, Inc.\n");
printf("Compiled on %s at %s\n", __DATE__, __TIME__);
```

Ogni volta che l'esecuzione ha inizio, il programma stampa due righe della forma

```
Wacky Windows (c) 2010 Wacky Software, Inc.
Compiled on Dec 23 2010 at 22:18:48
```

Questa informazione può essere utile per distinguere tra versioni diverse dello stesso programma.

Possiamo usare `_LINE_` e `_FILE_` per facilitare la localizzazione degli errori. Considerate il problema di identificare la posizione di una divisione per zero. Quando un programma C termina prematuramente a causa di una divisione per zero, solitamente

non c'è alcuna indicazione di quale divisione abbia causato il problema. La macro seguente può aiutare a definire con precisione la sorgente dell'errore:

```
#define CHECK_ZERO(divisor) \
 if (divisor == 0) \
 printf("!!! Attempt to divide by zero in line %d " \
 "of file %s !!!\n", __LINE__, __FILE__)
```

La macro CHECK\_ZERO verrebbe invocata prima di una divisione:

```
CHECK_ZERO(j);
k = i / j;
```

Nel caso in cui j fosse uguale a zero, verrebbe stampato un messaggio di questo tipo:

```
!!! Attempt to divide by zero in line 9 of file foo.c !!!
```

Le macro come questa che servono per la rilevazione degli errori sono piuttosto utili. Infatti la libreria C possiede una macro generale per la rilevazione degli errori chiamata assert [macro assert > 24.1].

La macro \_\_STDC\_\_ esiste e possiede il valore 1 nel caso in cui il compilatore è conforme allo standard C (sia C89 o C99). Dato che il preprocessore controlla questa macro, un programma può adattarsi a un compilatore che sia predatato rispetto al standard C89 (si veda la Sezione 14.4, per esempio).



## Macro predefinite aggiunte dal C99

Il C99 prevede alcune macro predefinite in più (Tabella 14.2).

**Tabella 14.2** Macro predefinite aggiunte dal C99

| Nome                      | Descrizione                                                                                |
|---------------------------|--------------------------------------------------------------------------------------------|
| __STDC_HOSTED__           | 1 se questa è un'implementazione hosted, 0 se freestanding                                 |
| __STDC_VERSION__          | Versione supportata dello standard C                                                       |
| __STDC_IEC_559__†         | 1 se è supportata l'aritmetica a virgola mobile IEC 60559                                  |
| __STDC_IEC_559_COMPLEX__† | 1 se è supportata l'aritmetica complessa IEC 60559                                         |
| __STDC_ISO_10646__†       | yyyymmL se i valori wchar_t sono conformi alle norme ISO 10646 dello specifico anno e mese |

†Definite condizionalmente

Per comprendere il significato della macro \_\_STDC\_HOSTED\_\_ abbiamo bisogno di un nuovo vocabolario. Un'implementazione del C è composta dal compilatore e da altro software necessario per eseguire i programmi. Il C99 suddivide le implementazioni in due categorie: **hosted** e **freestanding**. Un'implementazione **hosted** deve accettare qualsiasi programma conforme allo standard C99, mentre un'implementazione **freestanding** non deve necessariamente compilare i programmi che utilizzano



i tipi complessi [tipi complessi > 27.3] e gli header standard oltre a quelli basilari (in particolare un'implementazione freestanding non è obbligata a supportare l'header <stdio.h>). La macro `_STDC_HOSTED_` possiede il valore 1 se il compilatore è un'implementazione hosted, altrimenti possiede il valore 0.

La macro `_STDC_VERSION_` fornisce un modo per controllare quale versione dello standard C è riconosciuta dal compilatore. Questa macro apparve per la prima volta nell'*Amendment 1* (revisione 1) dello standard C89, dove il suo valore è stato specificato come la costante di tipo `long 199409L` (rappresentante l'anno e il mese della revisione dello standard). Se un compilatore è conforme allo standard C99, il valore è `199901L`. Per ogni versione successiva dello standard (e ogni revisione dello standard) questa macro assume un valore differente.

Un compilatore C99 può definire tre macro aggiuntive. Ogni macro è definita solo se il compilatore soddisfa certi requisiti.

- La macro `_STDC_IEC_599_` è definita (e ha il valore 1) se il compilatore esegue l'aritmetica a virgola mobile secondo lo standard IEC 60559 (un altro nome per lo standard IEEE 754 [standard floating point IEEE > 7.2]).
- La macro `_STDC_IEC_599_COMPLEX_` è definita (e ha il valore 1) se il compilatore esegue l'aritmetica complessa secondo lo standard IEC 60559.
- La macro `_STDC_ISO_10646_` è definita come una costante intera della forma `yyyymmL` (per esempio `199712L`) se i valori del tipo `wchar_t` [tipo wchar\_t > 25.2] sono rappresentati dai codici dello standard ISO/IEC 10646 [standard ISO/IEC 10646 > 25.2] (con le revisioni specificate dall'anno e dal mese).



## Argomenti delle macro vuoti

Il C99 permette che alcuni o tutti gli argomenti presenti nella chiamata di una macro possano essere vuoti. Una chiamata di questo tipo però conterrà lo stesso numero di virgolette di una chiamata normale (in questo modo è facile vedere quali argomenti sono stati omessi).

Nella maggior parte dei casi gli effetti di un argomento vuoto sono chiari. Qualunque sia il parametro corrispondente nell'elenco di sostituzione questo viene rimpiazzato dal nulla (scompare semplicemente dall'elenco di sostituzione). Ecco un esempio:

```
#define ADD(x,y) (x+y)
```

Dopo il preprocessamento, l'istruzione

```
i = ADD(j,k);
```

diventa

```
i = (j+k);
```

mentre l'istruzione

```
i = ADD(,k);
```

diventa

```
i = (+k);
```

Quando l'argomento vuoto è un operando degli operatori # o ##, si applicano delle regole speciali. Se un argomento vuoto viene reso una stringa dall'operatore #, il risultato è "" (la stringa vuota):

```
#define MK_STR(x) #x
-
char empty_string[] = MK_STR();
```

Dopo la fase di preprocessamento, l'istruzione si presenterà in questo modo:

```
char empty_string[] = "";
```

Se uno degli argomenti dell'operatore ## è vuoto, questo viene sostituito da un token segnaposto invisibile. Concatenare un token ordinario con un *token segnaposto* si traduce nel token originale (il segnaposto scompare). Se due segnaposto vengono concatenati, come risultato si ottiene un singolo token segnaposto. Una volta che l'espansione della macro è stata completata, i token segnaposto scompaiono tutti. Considerate l'esempio seguente:

```
#define JOIN(x,y,z) x##y##z
-
int JOIN(a,b,c), JOIN(a,,b,), JOIN(a,,c), JOIN(,,c);
```

Dopo il preprocessamento, la dichiarazione si presenterà in questo modo:

```
int abc, ab, ac, c;
```

Gli argomenti mancati vengono sostituiti con token segnaposto, che vanno a scomparire dopo essere stati concatenati con argomenti non vuoti. È possibile omettere anche tutti e tre gli argomenti della macro JOIN, il che condurrebbe a un risultato vuoto.

## C99 Macro con un numero variabile di argomenti

Nel C89 una macro deve avere un numero prefissato di argomenti. Il C99, invece, ammette macro che accettino un numero illimitato di argomenti [[elenco di argomenti a lunghezza variabile > 26.1](#)]. Questa caratteristica era disponibile già da diverso tempo per le funzioni e quindi non c'è da sorrendersi se alla fine anche le macro l'abbiano fatta propria.

La ragione principale nell'avere una macro con un numero variabile di argomenti è che essa possa passare questi ultimi a una funzione che ne accetta un numero variabile, come la printf o la scanf. Ecco un esempio:

```
#define TEST(condition, ...) { (condition)? \
 printf("Passed test: %s\n", #condition): \
 printf(_VA_ARGS_)}
```

Il token \_ , conosciuto come ellissi, viene posto alla fine dell'elenco dei parametri in modo da essere preceduto da quelli ordinari nel caso in cui ve ne fossero. La parola \_VA\_ARGS\_ è un identificatore speciale che può comparire solo in un elenco di sostituzione di una macro che abbia un numero variabile di argomenti. Infatti rappresenta tutti gli argomenti che corrispondono all'ellissi (deve esserci almeno un argomento corrispondente all'ellissi altrimenti l'identificatore è vuoto). La macro TEST richiede

almeno due argomenti. Il primo argomento è appaiato con il parametro condition, mentre i restanti argomenti corrispondono all'ellissi.

Ecco un esempio che mostra come può essere utilizzata la macro TEST:

```
TEST(voltage <= max_voltage,
 "Voltage %d exceeded %d\n", voltage, max_voltage);
```

Il preprocessore produrrà l'output seguente (riformattato per migliorare la leggibilità):

```
((voltage <= max_voltage)?
 printf("Passed test: %s\n", "voltage <= max_voltage"):
 printf("Voltage %d exceeds %d\n", voltage, max_voltage));
```

Quando il programma verrà eseguito, se voltage non è maggiore di max\_voltage allora verrà visualizzato il seguente messaggio:

```
Passed test: voltage <= max_voltage
```

In caso contrario il programma visualizzerà i valori di voltage e max\_voltage:

```
Voltage 125 exceeds 120
```

## C99 L'identificatore \_\_func\_\_

Un'altra caratteristica del C99 è quella dell'identificatore \_\_func\_\_, che non ha nulla a che fare con il preprocessore, tuttavia, come molte caratteristiche del preprocessore, è utile per le operazioni di debugging, per questo motivo ne parleremo qui.

Ogni funzione ha accesso all'identificatore \_\_func\_\_, il quale si comporta come una variabile stringa che contiene il nome della funzione correntemente in esecuzione. L'effetto è lo stesso che avremmo se ogni funzione contenesse la seguente dichiarazione all'inizio del suo corpo:

```
static const char __func__[] = "nome-funzione";
```

dove *nome-funzione* è il nome della funzione. L'esistenza di questo identificatore rende possibile la scrittura di macro per il debugging come quella seguente:

```
#define FUNCTION_CALLED() printf("%s called\n", __func__);
#define FUNCTION RETURNS() printf("%s returns\n", __func__);
```

Delle invocazioni a queste macro possono essere messe all'interno delle funzioni per tracciare le loro chiamate:

```
void f(void)
{
 FUNCTION_CALLED(); /* visualizza "f called" */
 ...
 FUNCTION RETURNS(); /* visualizza "f returns" */
}
```

Un altro utilizzo dell'identificatore \_\_func\_\_ è che questo può essere passato a una funzione per farle sapere il nome della funzione che l'ha invocata.

## 14.4 Compilazione condizionale

Il preprocessore del C riconosce un certo numero di direttive che supportano la **compilazione condizionale** (l'inclusione o l'esclusione di una sezione del testo del programma dipende dall'esito di un test eseguito dal preprocessore).

### Le direttive #if e #endif

Supponete di trovarvi durante la fase di debugging di un programma. Vorremmo che il programma stampasse il valore di certe variabili e per questo inseriamo delle chiamate alla printf in alcuni punti critici. Una volta localizzati i bachi, di solito è buona norma mantenere le chiamate alla printf per un possibile uso successivo. La compilazione condizionale ci permette di lasciare queste chiamate al loro posto facendo in modo che il compilatore le ignori.

Ecco come procederemo: per prima cosa definiremo una macro e le daremo un valore diverso da zero:

```
#define DEBUG 1
```

Il nome della macro non ha importanza. Successivamente circonderemo ogni gruppo di chiamate alla printf con una coppia #if-#endif:

```
#if DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif
```

Durante il preprocessamento la direttiva #if controlla il valore di DEBUG. Dato che il suo valore non è uguale a zero, il preprocessore lascia al loro posto le due chiamate alla printf (e quindi le due righe con #if e #endif scompaiono). Se cambiamo il valore della macro DEBUG ponendolo uguale a zero e ricompiliamo, allora il preprocessore rimuoverà quattro righe dal codice del programma. Il compilatore non vedrà le chiamate alla printf e quindi queste non occuperanno spazio all'interno del codice oggetto e nemmeno occuperanno tempo di esecuzione. Nel programma finale possiamo lasciare i blocchi #if-#endif permettendo così la produzione di informazioni di diagnostica (ricompilando con la macro DEBUG imposta a 1) se in un secondo momento si rivelassero necessarie.

In generale la direttiva #if ha il seguente formato:

#if espressione-costante

La direttiva #endif è anche più semplice

#endif



Quando il processore incontra la direttiva #if, calcola l'espressione costante. Se il valore dell'espressione è uguale a zero allora le righe comprese tra #if ed #endif verranno rimosse dal programma durante il preprocessamento. In caso contrario le righe comprese

tra le due direttive rimarranno nel programma e verranno elaborate dal compilatore (in questo caso #if ed #endif non avranno alcun effetto sul programma).

Vale la pena di notare che la direttiva #if tratta gli identificatori non definiti come delle macro con valore 0. Quindi se ci dimentichiamo di definire l'identificatore DEBUG, il test

```
#if DEBUG
fallirà (senza generare errori), mentre il test
#endif
avrà successo.
```

## L'operatore defined

Nella Sezione 14.3 abbiamo incontrato gli operatori # e ##. C'è solamente un altro operatore che è specifico per il preprocessore: l'operatore defined. Quando viene applicato a un identificatore, defined produce il valore 1 se l'identificatore è correntemente definito, mentre produce uno zero altrimenti. L'operatore defined viene normalmente utilizzato assieme alla direttiva #if permettendoci di scrivere

```
#if defined(DEBUG)
-
#endif
```

Le righe comprese tra #if ed #endif verranno incluse nel programma solo se DEBUG è stato definito come una macro. Le parentesi attorno a DEBUG non sono necessarie, infatti possiamo scrivere semplicemente

```
#if defined DEBUG
```

Dato che defined controlla solo se la macro DEBUG è definita o meno, non è necessario assegnare a quest'ultima un valore.

```
#define DEBUG
```

## Le direttive #ifdef e #ifndef

La direttiva #ifdef controlla se un identificatore è stato definito come una macro.

L'utilizzo di #ifdef è simile a quello della direttiva #if:

```
#ifdef identificatore
```

Righe che devono essere incluse se l'identificatore è definito come una macro  
#endif

**D&R** Effettivamente non c'è alcun bisogno della direttiva #ifdef visto che possiamo combinare la direttiva #if e l'operatore defined per ottenere lo stesso effetto. In altre parole, la direttiva

```
#ifdef identificatore
è equivalente a
#ifndef(identificatore)
```

La direttiva `#ifndef` è simile alla `#ifdef` ma controlla se l'identificatore *non* è stato definito come una macro:

**#ifndef identificatore**

Scrivere

```
#ifndef identificatore
equivale a scrivere
#ifndef defined(identificatore)
```

## Le direttive `#elif` e `#else`

I blocchi `#if`, `#ifdef` e `#ifndef` possono essere annidati proprio come le normali istruzioni `if`. Quando si applica l'annidamento è una buona idea utilizzare abbondantemente l'indentazione. Alcuni programmatore mettono un commento per ogni `#endif` di chiusura per indicare a quale condizione `#if` si riferisce:

```
#if DEBUG
-
#endif /* DEBUG */
```

Questa tecnica rende più facile per il lettore trovare l'inizio del blocco `#if`.

Per comodità il preprocessore supporta le direttive `#elif` ed `#else`:

**#elif espressione-costante**

**#else**

`#elif` ed `#else` possono essere utilizzate congiuntamente alle direttive `#if`, `#ifdef` o `#ifndef` per poter controllare una serie di condizioni:

```
#if espr1
```

*Righe che devono essere incluse se espr1 è diversa da zero*

```
#elif espr2
```

*Righe che devono essere incluse se espr1 è uguale a zero ma espr2 è diversa da zero*

```
#else
```

*Righe che devono essere incluse altrimenti*

```
#endif
```

Sebbene nell'esempio venga mostrata la direttiva `#if`, al suo posto possono essere utilizzate le direttive `#ifdef` e `#ifndef`. Tra `#if` ed `#endif` può comparire un numero qualsiasi di direttive `#elif` (ma al più una sola `#else`).

## Usi della compilazione condizionale

La compilazione condizionale è sicuramente adatta per il debugging ma il suo utilizzo non è ristretto solo a quel campo. Ecco alcune applicazioni comuni di questa tecnica.

- **Scrivere programmi che siano portabili su diverse macchine o sistemi operativi.** L'esempio seguente include uno dei tre gruppi di righe di codice a seconda che siano state definite le macro `WIN32`, `MAC_OS` o `LINUX`:

```
#if defined(WIN32)

-
#elif defined(MAC_OS)
-
#elif defined(LINUX)
-
#endif
```

Un programma può contenere diversi blocchi `#if` di questo tipo. All'inizio del programma deve essere definita una (e solo una) macro al fine di selezionare il tipo di sistema operativo. Per esempio, definendo la macro `LINUX` è possibile indicare al programma che verrà eseguito sul sistema operativo `LINUX`.

- **Scrivere programmi che possano essere compilati con compilatori differenti.** Compilatori diversi riconoscono versioni in qualche modo diverse del C. Alcuni seguono una versione standard del C mentre altri non lo fanno. Alcuni forniscono delle estensioni del linguaggio specifiche per la macchina, mentre altri non lo fanno o forniscono un diverso set di istruzioni. La compilazione condizionale può permettere a un programma di adattarsi ai diversi compilatori. Considerate il problema di scrivere un programma che debba essere compilato utilizzando un vecchio compilatore non standard. La macro `_STDC_` permette al preprocessore di capire se il compilatore è conforme o meno allo standard (C89 o C99). Se non lo fosse potremmo cambiare alcuni aspetti del programma. In particolare potremmo utilizzare il vecchio stile per la dichiarazione delle funzioni (discusso nella Sezione D&R alla fine del Capitolo 9) invece di utilizzare i prototipi di funzioni. In ogni punto dove avviene la dichiarazione di alcune funzioni, possiamo inserire le seguenti righe:

```
#if __STDC__
Prototipi delle funzioni
#else
Vecchio stile per le dichiarazioni delle funzioni
#endif
```

- **Fornire una funzione di default per una macro.** La compilazione condizionale permette di controllare se una macro è correntemente definita e, nel caso non lo fosse, di assegnarle un valore di default. Per esempio, le righe seguenti definiscono la macro `BUFFER_SIZE` nel caso in cui questa non fosse già stata definita.

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

- **Disabilitare temporaneamente il codice che contiene dei commenti**  
Non possiamo utilizzare `/* */` per aggiungere indicatori di commento al codice che già ne contiene in quello stile. Possiamo invece utilizzare la direttiva `#if`:

```
#if 0
Righe contenenti dei commenti
#endif
```



Disabilitare del codice in questo modo spesso viene chiamato “condizionamento”. La Sezione 15.2 discute un altro utilizzo comune della compilazione condizionale: proteggere i file di header dalle inclusioni multiple.

## 14.5 Direttive varie

Dedichiamo la parte finale di questo capitolo alle direttive `#error`, `#line` e `#pragma`, che sono più specializzate di quelle che abbiamo già esaminato e vengono utilizzate con minore frequenza.

### La direttiva `#error`

La direttiva `#error` segue il formato

```
#error messaggio
```

dove *messaggio* è una qualsiasi sequenza di token. Se il preprocessore incontra una direttiva `#error`, stampa un messaggio di errore che deve includere al suo interno la frase *messaggio*. L'esatto formato del messaggio di errore può variare da un compilatore all'altro, può essere qualcosa come

Error directive: messaggio

o semplicemente

`#error messaggio`

Incontrare una direttiva `#error` indica che nel programma è presente un difetto piuttosto serio. Alcuni compilatori terminano immediatamente la compilazione senza cercare di individuare altri errori.

Le direttive `#error` sono utilizzate frequentemente in modo congiunto alla compilazione condizionale per controllare che non si verifichino eventuali problemi durante la normale compilazione. Supponete per esempio di volervi assicurare che un programma non possa essere compilato su una macchina dove il tipo `int` non è in grado di contenere numeri fino a 100000. Il più grande valore `int` è rappresentato dalla macro `INT_MAX` [macro `INT_MAX > 23.2`] e, quindi, tutto quello di cui abbiamo bisogno è di invocare la direttiva `#error` nel caso in cui `INT_MAX` sia minore di 100000:

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

Cercare di compilare il programma su una macchina i cui interi sono memorizzati su 16 bit produce un messaggio come

Error directive: int type is too small

La direttiva `#error` si trova spesso nella parte `#else` di una serie `#if-#elif-#else:`

```
#if defined(WIN32)
-
#elif defined(MAC_OS)
-
#elif defined(LINUX)
-
#else
#error No operating system specified
#endif
```

## La direttiva `#line`

La direttiva `#line` viene utilizzata per alterare il modo in cui vengono numerate le righe di un programma (le righe di solito sono numerate come 1, 2, 3 e così via). Possiamo utilizzare questa direttiva per far credere al compilatore che sta leggendo il programma da un file con nome diverso.

La direttiva `#line` ha due formati. Nel primo formato viene specificato il numero della riga:



`#line n`

*n* deve corrispondere a una sequenza di cifre rappresentanti un numero intero compreso tra 1 e 32767 (2147483647 nel C99). Questa direttiva fa sì che le linee seguenti del programma vengano numerate come *n*+1, *n*+2 e così via.

Nel secondo formato della direttiva `#line` vengono specificati sia il numero della riga che il nome del file:



`#line n "file"`

In questo modo il compilatore suppone che le righe seguenti a questa direttiva provengano dal file *file* con numeri che cominciano a partire da *n*. I valori di *n* e/o della stringa *file* possono essere specificati utilizzando delle macro.

Un effetto della direttiva `#line` è quello di modificare il valore della macro `_LINE_` (ed eventualmente anche quello della macro `_FILE_`). Cosa ancora più importante è che la maggior parte dei compilatori utilizza le informazioni della direttiva `#line` quando generano i messaggi di errore.

Supponete per esempio che la seguente direttiva compaia all'inizio del file *foo.c*:

`#line 10 "bar.c"`

Assumiamo che il compilatore abbia trovato un errore nella riga 5 del file *foo.c*. Il messaggio di errore del compilatore farà riferimento alla riga 13 del file *bar.c*, non alla riga 5 del file *foo.c* (perché la riga 13? La direttiva occupa la riga 1 di *bar.c* e quindi la nuova numerazione del file comincia alla riga 2 che viene trattata come la riga 10 del file *bar.c*).

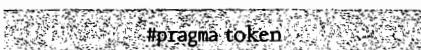
A prima vista la direttiva `#line` può confondere. Perché dovremmo volere messaggi di errore si riferiscono a righe diverse e a file diversi? Questo non rende più difficile il debug dei programmi?

Infatti la direttiva `#line` non viene utilizzata spesso dai programmatore; viene utilizzata principalmente dai programmi che generano del codice C come loro output. Il più famoso esempio di questo tipo di programmi è *yacc* (*Yet Another Compiler*), un'utility di UNIX che genera automaticamente parti di un compilatore (la versione GNU di yacc è chiamata *bison*). Prima di utilizzare yacc i programmatore preparano un file contenente sia informazioni utili a yacc sia frammenti di codice. A partire da questo file l'utility yacc genera un programma C (*y.tab.c*) che incarna il codice fornito dal programmatore. Il programmatore poi compila *y.tab.c* nel modo normale. Inserendo delle direttive `#line`, yacc inganna il compilatore facendogli credere che il codice provenga dal file originale (quello scritto dal programmatore). Come risultato si ha che un qualsiasi messaggio di errore prodotto durante la compilazione di *y.tab.c* si riferisce alle righe del file originario e non a quelle di *y.tab.c*. Questo rende il debugging più facile perché i messaggi di errore fanno riferimento al codice scritto dal programmatore e non a quello generato da yacc (che è più complesso).

## La direttiva `#pragma`

La direttiva `#pragma` fornisce un modo per richiedere un comportamento specifico della parte del compilatore. Questa direttiva è utile principalmente per i programmi che sono insolitamente grandi o che hanno bisogno di sfruttare alcune particolari capacità del compilatore.

La direttiva `#pragma` segue il formato



dove *token* è un elenco arbitrario di token. Questa direttiva può essere molto semplice (un singolo token) o molto elaborata:

```
#pragma data(heap_size => 1000, stack_size => 2000)
```

Non ci deve sorprendere il fatto che l'insieme di comandi che possono comparire nelle direttive `#pragma` sia diverso da un compilatore all'altro. Dovete consultare la documentazione del vostro compilatore per vedere quali comandi sono ammessi e cosa questi facciano. Tra l'altro il preprocessore deve ignorare qualsiasi direttiva `#pragma` contenente un comando non riconosciuto e non è permessa la generazione di un messaggio di errore.

Nel C89 non sono presenti dei comandi `pragma standard` (vengono tutti definiti dall'implementazione). Il C99 possiede tre comandi standard e tutti utilizzano

come il primo dei token che seguono `#pragma`. Questi comandi sono `FP_CONTEXT` (trattato nella Sezione 23.4), `CX_LIMITED_RANGE` (Sezione 27.4) e `FENV_ACCESS` (Sezione 27.6).

C99

## L'operatore `_Pragma`

Il C99 introduce l'operatore `_Pragma` che viene utilizzato congiuntamente alla direttiva `#pragma`. Un'espressione `_Pragma` ha il formato

`_Pragma( "stringa-letterale" )`

Quando il preprocessore incontra un'espressione di questo tipo trasforma la stringa letterale (il termine utilizzato dallo standard è *destringize*) rimuovendo i doppi apici attorno alla stringa e sostituendo le sequenze di escape `\"` e `\\` rispettivamente per i caratteri `"` e `\`. Il risultato è una serie di token che sono trattati come se appartenessero a una direttiva `#pragma`. Per esempio, scrivere

`_Pragma("data(heap_size => 1000, stack_size => 2000)")`

equivale a scrivere

`#pragma data(heap_size => 1000, stack_size => 2000)`

L'operatore `_Pragma` ci permette di aggirare una limitazione del preprocessore vero il fatto che le direttive di preprocessamento non possano generare un'altra direttiva. Il `_Pragma` invece non è una direttiva ma un operatore e quindi può comparire all'interno della definizione di una macro. Questo permette all'espansione di una macro di lasciarsi dietro una direttiva `#pragma`.

Esaminiamo un esempio preso dal manuale di GCC. La seguente macro utilizza l'operatore `_Pragma`:

```
#define DO_PRAGMA(x) _Pragma(#x)
```

La macro viene invocata in questo modo:

```
DO_PRAGMA(GCC dependency "parse.y")
```

Il risultato ottenuto dall'espansione è

```
#pragma GCC dependency "parse.y"
```

che è uno dei comandi `pragma` supportati da GCC (il comando genera un warning nel caso in cui la data del file specificato (`parse.y` nel nostro esempio) è più recente della data del file corrente, ovvero di quello che è in compilazione). Osserviamo che l'argomento della chiamata a `DO_PRAGMA` è una serie di token. L'operatore `#` presente nella definizione di `DO_PRAGMA` fa sì che i token formino la stringa `"GCC dependency \"parse.y\""`. Questa stringa viene passata all'operatore `_Pragma`, il quale la distribuisce producendo una direttiva `#pragma` contenente i token originali.

## Domande & Risposte

**D:** Abbiamo visto programmi contenenti un operatore # su una riga a sé stante. Questo è ammissibile?

**R:** Sì. Questa è la direttiva nulla che non ha alcun effetto. Alcuni programmatori utilizzano direttive nulle per distanziare all'interno dei blocchi di compilazione condizionale:

```
#if INT_MAX < 100000
#
#error int type is too small
#
#endif
```

Delle righe vuote funzionerebbero ugualmente ma il carattere # aiuta il lettore a capire l'estensione del blocco.

**D:** Non siamo sicuri di quali costanti debbano essere definite come macro. Ci sono delle linee guida da seguire? [p. 331]

**R:** Una regola empirica dice che ogni costante numerica diversa da 0 e 1 debba essere dichiarata come una costante simbolica. I caratteri e le stringhe costanti sono problematici visto che sostituirli con una macro non sempre migliora la leggibilità. Utilizzare una macro al posto di un carattere o di una stringa costante va bene se (1) la costante viene utilizzata più di una volta e (2) c'è la possibilità che la costante venga modificata un giorno. Per seguire la regola (2) non utilizzeremo delle macro come

```
#define NULL '\0'
```

sebbene alcuni programmatori lo facciano.

**D:** Cosa fa l'operatore # se l'argomento che deve essere trasformato in una stringa contiene un carattere " o un \ ? [p.326]

**R:** L'operatore converte il carattere " in \" e il carattere \ in \\. Considerate la seguente macro:

```
#define STRINGIZE(x) #x
```

Il preprocessore sostituirà STRINGIZE("foo") con "\"foo\\\"".

**\*D:** Non riusciamo a far funzionare correttamente la seguente macro:

```
#define CONCAT(x,y) x##y
```

**CONCAT(a,b)** restituisce ab come ci si aspettava, ma **CONCAT(a, CONCAT(b,c))** restituisce uno strano risultato. Cosa sta succedendo?

**R:** Grazie a delle regole che Kernighan e Ritchie chiamavano "bizzarre", le macro il cui elenco di sostituzione dipende dall'operatore ##, di solito non possono essere chiamate in forma annidata. Il problema è che CONCAT(a, CONCAT(b,c)) non viene espanso nel modo "normale", ovvero con CONCAT(b,c) che restituisce bc e poi con CONCAT(a, bc) che restituisce abc. I parametri di una macro che in un elenco di sostituzione sono preceduti o seguiti da ## non vengono espansi nel momento della sostituzione. Come

risultato si ha che `CONCAT(a, CONCAT(b,c))` viene espanso in `aCONCAT(b,c)` che non può essere espanso ulteriormente visto che non c'è alcuna macro chiamata `aCONCAT`.

C'è un modo per risolvere il problema ma non è molto elegante. Il trucco è quello di definire una seconda macro che semplicemente chiama la prima:

```
#define CONCAT2(x,y) CONCAT(x,y)
```

Scrivendo `CONCAT2(a, CONCAT2(b,c))` si ottiene il risultato voluto. Quando il preprocessore espande la chiamata esterna a `CONCAT2`, espanderà anche quella interna. La differenza questa volta è che l'elenco di sostituzione di `CONCAT2` non contiene l'operatore `##`. Se tutto questo vi sembra non avere senso non preoccupatevi, questo non è un problema che si verifica spesso.

L'operatore `#` presenta una difficoltà simile. Se in un elenco di sostituzione compare un `#x`, dove `x` è un parametro della macro, allora l'argomento corrispondente non viene espanso. Quindi se `N` è una macro che rappresenta la costante 10 e `STR(x)` possiede `#x` come elenco di sostituzione, allora l'espansione di `STR(N)` restituisce "N" e non "10". La soluzione è simile a quella utilizzata con `CONCAT`: definire una seconda macro il cui scopo sia quello di chiamare la `STR`.

**\*D:** Supponiamo che il preprocessore incontri il nome originale della macro durante una successiva scansione, così come capita nell'esempio seguente:

```
#define N (2*M)
#define M (N+1)

i = N; /* ciclo infinito? */
```

**Il preprocessore sostituirà N con (2\*M) e poi sostituirà M con (N+1). Il preprocessore sostituirà nuovamente N entrando in un ciclo infinito? [p.338]**

**R:** Alcuni vecchi preprocessori entrerebbero in un ciclo infinito mentre quelli più recenti no. Secondo lo standard C, se il nome originale della macro ricompare durante l'espansione di un'altra macro, allora il nome non viene sostituito. Ecco come si presenterebbe l'assegnamento dopo il preprocessamento:

```
i = (2*(N+1));
```

Alcuni programmati intraprendenti sfruttano questo comportamento scrivendo delle macro con nomi che combaciano con parole riservate o con delle funzioni della libreria standard. Considerate la funzione di libreria `sqrt` [funzione `sqrt` > 23.3] che calcola la radice quadrata dei suoi argomenti restituendo un valore dipendente dall'implementazione nel caso in cui l'argomento fosse negativo. Forse vorremmo che la `sqrt` restituisse 0 con argomenti negativi. Poiché la `sqrt` fa parte della libreria standard non possiamo modificarla facilmente. Possiamo però definire una macro `sqrt` che restituisce 0 quando le viene passato un argomento negativo:

```
#undef sqrt
#define sqrt(x) ((x)>=0?sqrt(x):0)
```

Una successiva chiamata alla `sqrt` verrebbe intercettata dal preprocessore il quale la espanderebbe nella espressione condizionale mostrata qui. La chiamata alla `sqrt`

contenuta all'interno dell'espressione condizionale non verrebbe sostituita durante la successiva scansione del preprocessore e quindi verrebbe gestita dal compilatore. Osservate l'utilizzo di `#undef` prima della definizione di `sqrt` come macro. Come vedremo nella Sezione 21.1, alla libreria standard è permesso avere sia una macro che una funzione con lo stesso nome. Annullare la definizione di `sqrt` prima di definire la nostra macro è una misura cautelativa nel caso in cui la libreria avesse già definito una macro `sqrt`.

**D: Ottengo un errore quando provo a utilizzare delle macro predefinite come `_LINE_` e `_FILE_`. C'è bisogno di includere un particolare header per poterlo fare?**

**R:** No. Queste macro vengono riconosciute automaticamente dal preprocessore. Assicuratevi di aver scritto *due underscore* all'inizio e alla fine del nome di ogni macro e non uno soltanto.

**D: Qual è lo scopo della distinzione tra "hosted implementation" e "free-standing implementation"? Se una implementazione freestanding non porta nemmeno l'header `<stdio.h>` qual è il suo scopo? [p. 342]**

**R:** Un'implementazione *hosted* è necessaria per la maggior parte dei programmi (inclusi quelli presenti in questo libro), i quali si basano sul sistema operativo sottostante per l'input/output e gli altri servizi essenziali. Un'implementazione *freestanding* del C potrebbe essere utilizzata da quei programmi che non richiedono un sistema operativo (o solo un sistema operativo minimale). Per esempio, sarebbe necessaria un'implementazione *freestanding* per scrivere il *kernel* di un sistema operativo (il quale non richiede dell'input/output tradizionale e quindi non ha bisogno di `<stdio.h>`). Le implementazioni *freestanding* sono utili anche per la scrittura di software per i sistemi *embedded*.

**D: Pensavo che il preprocessore fosse semplicemente un editor. Come fa a calcolare le espressioni costanti? [p. 346]**

**R:** Il preprocessore è più sofisticato di quello che potreste aspettarvi, conosce abbastanza C da essere in grado di calcolare delle espressioni costanti, sebbene non lo faccia allo stesso modo del compilatore (per esempio il preprocessore tratta ogni nome non definito come se possedesse il valore 0. Le altre differenze sono troppo "esoteriche" per essere discusse qui). Nella pratica gli operandi di un'espressione costante del preprocessore solitamente sono costanti, macro che rappresentano costanti e usi dell'operatore `defined`.

**D: Perché il C fornisce le direttive `#ifdef` e `#ifndef` dato che possiamo ottenere lo stesso effetto utilizzando la direttiva `#if` e l'operatore `defined?` [p. 347]**

**R:** Le direttive `#ifdef` e `#ifndef` fanno parte del C sin dal 1970. L'operatore `defined`, d'altro canto, è stato aggiunto al C nel 1980 durante la standardizzazione. Di conseguenza la domanda giusta è: perché l'operatore `defined` è stato aggiunto al linguaggio? La risposta è che `defined` incrementa la flessibilità. Invece di controllare l'esistenza di una singola macro utilizzando `#ifdef` o `#ifndef`, ora possiamo controllare un qualsiasi numero di macro utilizzando `#if` assieme a `defined`. Per esempio, la seguente direttiva controlla se `FOO` e `BAR` sono definite mentre `BAZ` non lo è:

```
#if defined(FOO) && defined(BAR) && !defined(BAZ)
```

D: Volevamo compilare un programma di cui non avevamo terminato la scrittura e per questo abbiamo "reso condizionale" la parte non terminata:

```
#if 0
```

```
#endif
```

Al momento della compilazione del programma ci è stato restituito un messaggio di errore che faceva riferimento a una delle righe comprese tra #if ed #endif. Il preprocessore non doveva semplicemente ignorare queste righe? [p. 350]

R: No, le righe non vengono completamente ignorate. I commenti vengono elaborati prima che le direttive del preprocessore vengano eseguite e il codice sorgente viene suddiviso in token per il preprocessamento. Quindi un commento non terminato presente tra #if ed #endif può essere causa di un messaggio di errore. Anche un apice o doppio apice non accoppiato può provocare un comportamento indefinito.

## Esercizi

### Sezione 14.3

- Scrivete una macro parametrica che calcoli i seguenti valori:

- (a) Il cubo di x.
- (b) Il resto ottenuto dividendo n per 4.
- (c) 1 se il prodotto di x e y è minore di 100, 0 altrimenti.

Le vostre macro funzionano sempre? Descrivete quali argomenti non le farebbero funzionare.

- W 2. Scrivete la macro NELEMS(a) che calcola il numero di elementi presenti nel vettore unidimensionale a. Suggerimento: guardate la discussione sull'operatore sizeof nella Sezione 8.1.
- 3. Sia DOUBLE la seguente macro:

```
#define DOUBLE(x) 2*x
```

- (a) Qual è il valore di DOUBLE(1+2)?
- (b) Qual è il valore di 4/DDOUBLE(2)?
- (c) Correggete la definizione di DOUBLE.

- W 4. Per ognuna delle seguenti macro fornite un esempio che illustri un problema che si potrebbe verificare con la macro stessa e fornite la soluzione.
  - (a) #define AVG(x,y) (x+y)/2
  - (b) #define AREA(x,y) (x)\*(y)

- W 5. \*Sia TOUPPER la seguente macro:

```
#define TOUPPER ('a'<=(c)&&(c)<='z'? (c)-'a'+'A':(c))
```

Sia s una stringa e sia i una variabile int. Mostrate l'output prodotto da ognuno dei seguenti frammenti di programma.

- (a) strcpy(s, "abcd");  
i = 0;

```

putchar(TOUPPER(s[++i]));
(b) strcpy(s, "0123");
 i = 0;
 putchar(TOUPPER(s[++i]));

```

6. (a) Scrivete la macro DISP(f, x) che si espande in una chiamata alla printf che visualizza il valore della funzione f quando viene chiamata con l'argomento x.
- Per esempio:

```

DISP(sqrt, 3.0);
deve espandersi in
printf("sqrt(%g) = %g\n", 3.0, sqrt(3.0));

```

- (b) Scrivete la macro DISP2(f,x,y), questa è simile alla macro DISP ma lavora con funzioni a due argomenti.

7. \*Sia GENERIC\_MAX una macro di questo tipo:

```

#define GENERIC_MAX(type) \
type type##_max(type x, type y) \
{ \
return x > y ? x : y; \
}

```

- (a) Mostrate l'espansione eseguita dal preprocessore su GENERIC\_MAX(long).
- (b) Spiegate perché GENERIC\_MAX non funziona con tipi base come unsigned long.
- (c) Descrivete una tecnica che permetterebbe di utilizzare GENERIC\_MAX con tipi base come unsigned long. *Suggerimento:* non modificate la definizione di GENERIC\_MAX.

8. \*Supponete di voler scrivere una macro che si espanda in una stringa contenente il numero della riga e del file correnti. In altre parole vorremmo scrivere

```
const char *str = LINE_FILE;
```

per ottenere l'espansione

```
const char *str = "Line 10 of file foo.c";
```

dove foo.c è il file contenente il programma mentre 10 è la riga nella quale compare l'invocazione alla LINE\_FILE. *Attenzione:* questo esercizio è solo per esperti. Assicuratevi di aver letto attentamente la sezione D&R prima di tentare!

9. Scrivete le seguenti macro parametriche.

- (a) CHECK(x,y,n) – Ha il valore 1 se sia x che y sono compresi tra 0 e n-1, estremi inclusi.
- (b) MEDIAN(x,y,z) – Cerca la mediana di x, y e z.
- (c) POLINOMIAL(x) – Calcola il polinomio  $3x^5 + 2x^4 - 5x^3 - x^2 + 7x - 6$ .

10. Spesso (ma non sempre) le funzioni possono essere scritte come macro parametriche. Discutete quali caratteristiche debba avere una funzione affinché questa non sia implementabile come una macro.

11. (C99) I programmati C usano spesso la funzione `fprintf` per scrivere dei messaggi di errore:

```
fprintf [funzione fprintf > 22.3] (stderr, "Range error: index = %d\n", index);
stderr [stream stderr > 22.1] è lo stream di standard error del C. I restanti argomenti sono gli stessi della printf, a partire dalla stringa di formato. Scrivete una macro chiamata ERROR che generi la chiamata alla fprintf mostrata quando le vengono passati una stringa di formato e gli oggetti che devono essere visualizzati:

ERROR("Range error: index = %d\n", index);
```

- Sezione 14.4** 12. Supponete che la macro `M` sia definita in questo modo:

W

```
#define M 10
```

Quale dei seguenti test darà esito negativo?

- (a) #if M
- (b) #ifdef M
- (c) #ifndef M
- (d) #if defined(M)
- (e) #if !defined(M)

13. (a) Mostrate come si presenterà il seguente programma dopo il preprocessamento. Potete ignorare ogni riga aggiunta al programma come risultato dell'inclusione dell'header `<stdio.h>`.

```
#include <stdio.h>

#define N 100

void f(void);

int main(void)
{
 f();
#ifndef N
#define N
#endif
 return 0;
}
void f(void)
{
#if defined(N)
 printf("N is %d\n", N);
#else
 printf("N is undefined\n");
#endif
}
```

- (b) Quale sarà l'output del programma?

- W 14. \* Mostrate come si presenterà il seguente programma dopo il preprocessamento. Alcune righe del programma possono causare degli errori di compilazione, trovateli.

```

#define N = 10
#define INC(x) x+1
#define SUB(x,y) x-y
#define SQR(x) ((x)*(x))
#define CUBE(x) (SQR(x)*(x))
#define M1(x,y) x##y
#define M2(x,y) #x #y

int main(void)
{
 int a[N], i, j, k, m;
#define N
 i = j;
#else
 j = i;
#endif
 i = 10 * INC(j);
 i = SUB(j, k);
 i = SQR(SQR(j));
 i = CUBE(j);
 i = M1(j,k);
 puts(M2(i, j));

#undef SQR
 i = SQR(j);
#define SQR
 i = SQR(j);

 return 0;
}

```

15. Supponete che un programma debba visualizzare un messaggio in inglese, francese o spagnolo. Utilizzando la compilazione condizionale, scrivete un frammento di programma che visualizzi uno dei tre messaggi seguenti a seconda che la specifica macro sia definita o meno:

Insert Disk 1      (se la macro ENGLISH è definita)  
 Inserez Le Disque 1    (se la macro FRENCH è definita)  
 Inserte El Disco 1    (se la macro SPANISH è definita)

#### Sezione 14.5

C99

16. \*Assumete che siano effettive le seguenti definizioni di macro:

```
#define IDENT(x) PRAGMA(ident #x)
#define PRAGMA(x) _Pragma(#x)
```

Come si presenterà la riga seguente dopo l'espansione della macro?

IDENT(foo)

# 15 Scrivere programmi di grandi dimensioni

Sebbene alcuni programmi C siano sufficientemente brevi da essere posti in un singolo file, la maggior parte non lo sono. Programmi che sono costituiti da più di un file sono la regola e non l'eccezione. In questo capitolo vedremo che un tipico programma è costituito da diversi file sorgente e tipicamente anche da alcuni file header. I file sorgente contengono le definizioni delle funzioni e delle variabili esterne. I file header contengono le informazioni che devono essere condivise tra i file sorgente. La Sezione 15.1 parla dei file sorgente, mentre la Sezione 15.2 tratta i file header. La Sezione 15.3 descrive come dividere il programma in file sorgente e file header. Successivamente la Sezione 15.4 fa vedere come "fare il build" (compilare e fare il linking) di un programma che consiste di più file. Tale sezione illustra anche come rieseguire il build del programma dopo che una parte di questo è stata modificata.

## 15.1 File sorgente

Fino a questo momento abbiamo assunto che un programma C sia costituito da un singolo file. In realtà un programma può essere diviso su un qualsiasi numero di file sorgente. Per convenzione i file sorgente hanno estensione .c. Ogni file sorgente contiene parte del programma, principalmente definizioni di funzioni e variabili. Un file sorgente deve contenere una funzione chiamata `main` che fa da punto di partenza per il programma.

Supponete per esempio di scrivere un semplice programma calcolatrice che calcoli espressioni intere immesse nella notazione polacca inversa (RPN) nella quale gli operatori seguono gli operandi. Se l'utente immette un'espressione come

30 5 - 7 \*

vogliamo che il programma stampi il suo valore (175 in questo caso). Calcolare un'espressione RPN è facile se facciamo in modo che il programma legga gli operandi e gli operatori uno alla volta utilizzando uno stack [stack > 10.2] per tenere traccia dei risultati intermedi. Se il programma legge un numero dobbiamo metterlo nello stack. Se invece legge un operatore dobbiamo effettuare il pop di due numeri dallo stack, effettuare l'operazione e rimettere il risultato nello stack. Quando un programma raggiunge la fine dell'input immesso dall'utente, il valore dell'espressione

si trova nello stack. Per esempio, il programma calcolerà l'espressione  $30 \cdot 5 - 7 * 1$

- modo seguente:

1. inserimento di 30 nello stack;
2. inserimento di 5 nello stack;
3. estrazione dei due numeri presenti in cima allo stack, sottrazione di 5 da inserimento del risultato (25) nello stack;
4. inserimento di 7 nello stack;
5. estrazione dei due numeri presenti in cima allo stack, moltiplicazione di quattro inserimento del risultato nello stack.

Dopo questi passi lo stack conterrà il valore dell'espressione (175).

Tramutare questo procedimento in un programma non è difficile. La funzione del programma conterrà un ciclo che eseguirà le seguenti azioni:

- leggere un "token" (un numero o un operatore);
- se il token è un numero, inserimento di questo nello stack;
- se il token è un operatore, estrarre dallo stack i suoi operandi, eseguire l'operazione e inserire il risultato nello stack.

Quando un programma come questo viene suddiviso su più file, ha senso inserire all'interno dello stesso file le funzioni e le variabili collegate. La funzione che legge token può andare in un file sorgente (diciamo `token.c`), assieme a tutte le funzioni che hanno a che fare con i token. Le funzioni relative allo stack come `push`, `pop`, `make_token`, `is_empty` e `is_full` andranno in un file diverso, che chiameremo `stack.c`. Anche le variabili rappresentanti lo stack potranno andare all'interno di `stack.c`. La funzione `main` verrà messa in un file ancora differente che chiameremo `calc.c`.

Suddividere un programma in più file sorgente presenta vantaggi significativi:

- raggruppare funzioni e variabili collegate all'interno di un singolo file aiuta a rendere chiara la struttura del programma;
- ogni file sorgente può essere compilato separatamente (un grosso risparmio di tempo se il programma è grande e deve essere modificato di frequente, una cosa piuttosto comune durante lo sviluppo);
- le funzioni diventano facilmente riutilizzabili in altri programmi quando vengono raggruppate in file sorgente separati. Nel nostro esempio, suddividere `stack.c` e `token.c` dalla funzione `main` semplifica un futuro riutilizzo delle funzioni dello stack e quelle relative ai token.

## 15.2 File header

Quando suddividiamo un programma in diversi file sorgente si presentano dei problemi: come fa una funzione di un file a chiamare una funzione che è stata definita in un altro file? Come fa una funzione ad accedere a una variabile esterna presente in un altro file? Come fanno due file a condividere la stessa definizione di macro di tipo? La risposta risiede nella direttiva `#include`, che rende possibile la condivisione delle informazioni (prototipi delle funzioni, definizioni delle macro, definizioni di tipi e altro) tra i file sorgente.

La direttiva `#include` dice al processore di aprire uno specifico file e di inserire il suo contenuto all'interno del file corrente. Quindi, se vogliamo che diversi file sorgente abbiano accesso alla stessa informazione, mettiamo questa informazione in un file e poi utilizziamo la direttiva `#include` per inserire il contenuto di questo all'interno di ogni file sorgente. I file che vengono inclusi in questo modo vengono chiamati file header (o file include), li tratteremo in maggiore dettaglio in una sezione a venire. Per convenzione i file header hanno estensione `.h`.

Nota: lo standard C utilizza il termine "file sorgente" per fare riferimento a tutti i file scritti dal programmatore, sia i file `.c` che quelli `.h`. Noi utilizzeremo il termine "file sorgente" solo per riferirci ai file `.c`.

## La direttiva `#include`

La direttiva `#include` possiede principalmente due formati. Il primo viene utilizzato per i file header che appartengono alla stessa libreria del C:

(formato 1) `#include <nomefile>`

Il secondo formato viene utilizzato per tutti gli altri file header, inclusi quelli scritti da noi stessi:

(formato 2) `#include "nomefile"`



La differenza tra i due è sottile dato che ha a che fare con il modo nel quale il compilatore cerca i file header. Ecco le regole seguite dalla maggior parte dei compilatori:

- `#include <nomefile>`: cerca nella (o nelle) cartella(e) dove risiedono i file header di sistema (nei sistemi UNIX, per esempio, i file header di sistema solitamente vengono conservati nella directory `/usr/include`);
- `#include "nomefile"`: cerca i file nella directory corrente e poi nella (o nelle) directory dove risiedono i file header di sistema.

Soltanmente i percorsi nei quali i file header vengono cercati possono essere modificati, spesso con un'opzione della riga di comando come `-Ipath`.



Non utilizzate le parentesi acute quando includete dei file header che avete scritto personalmente:

```
#include <myheader.h> /*** SBAGLIATO ***/
```

Il preprocessore probabilmente andrà alla ricerca di `myheader.h` nel luogo dove vengono tenuti i file header di sistema (e naturalmente non lo troverà).

Il nome del file inserito in una direttiva `#include` può contenere delle informazioni che aiutino a localizzare il file stesso, come il percorso o l'indicatore del drive:

```
#include "c:\cprogs\utils.h" /* Windows path */
#include "/cprogs/utils.h" /* UNIX path */
```

Sebbene i doppi apici presenti nella direttiva `#include` facciano sì che il nome del file sembri una stringa letterale, il preprocessore non lo tratta in quel modo (questa è una fortuna visto che \c e \u – che compaiono nell'esempio Windows – vengono trattati come sequenze di escape nelle stringhe letterali).

#### PORATABILITÀ

Di solito è meglio non includere nelle direttive `#include` delle informazioni sul percorso e sul drive. Questo tipo di informazioni rende difficile la compilazione di un programma quando questo viene trasportato da una macchina a un'altra o, peggio, quando viene trasportato da un sistema operativo a un altro.

Per esempio, le seguenti direttive `#include` per un sistema Windows specificano delle informazioni sul drive e/o sul percorso che possono non essere sempre valide:

```
#include "d:utils.h"
#include "\cprogs\include\utils.h"
#include "d:\cprogs\include\utils.h"
```

Le direttive seguenti sono migliori: non specificano il drive e il percorso è relativo e non assoluto.

```
#include "utils.h"
#include "..\include\utils.h"
```

La direttiva `#include` possiede un terzo formato che viene utilizzato più raramente rispetto ai due già visti:

(formato 3) `#include tokens`

dove `tokens` è una qualsiasi sequenza di token del preprocessore [token del preprocessore > 14.3]. Il preprocessore analizzerà i token e sostituirà ogni macro che incontra. Dopo la sostituzione delle macro, la direttiva risultante deve corrispondere a una delle altre forme di `#include`. Il vantaggio del terzo tipo di `#include` è che il nome del file può essere definito da una macro invece che essere pre-codificato nella direttiva stessa così come mostra il seguente esempio:

```
#if defined(IA32)
 #define CPU_FILE "ia32.h"
#elif defined(IA64)
 #define CPU_FILE "ia64.h"
#elif defined(AMD64)
 #define CPU_FILE "amd64.h"
#endif

#include CPU_FILE
```

## Condividere le definizioni delle macro e le definizioni dei tipi

La maggior parte dei grandi programmi contengono delle definizioni di macro e delle definizioni di tipo che hanno bisogno di essere condivise tra diversi file sorgente (o, nella maggior parte dei casi, da tutti i file sorgente). Queste definizioni dovrebbero essere inserite nei file header.

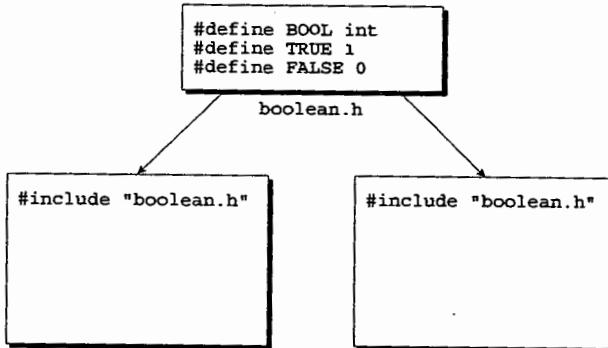
Per esempio, supponete di scrivere un programma che utilizzi delle macro chiamate BOOL, TRUE e FALSE (naturalmente nel C99 non c'è bisogno di queste macro perché l'header `<stdbool.h>` ne definisce di simili). Invece di ripetere la definizione di queste macro in ogni file sorgente che ne avesse bisogno, ha più senso inserire la definizione in un file header con un nome come `boolean.h`:

```
#define BOOL int
#define TRUE 1
#define FALSE 0
```

Qualsiasi file sorgente che avesse bisogno di queste macro potrebbe contenere semplicemente

```
#include "boolean.h"
```

Nella figura seguente i due file includono `boolean.h`:



Anche le definizioni di tipo sono comuni nei file header. Per esempio, invece di definire una macro BOOL, potremmo utilizzare `typedef` per creare un tipo `Bool`. Se facessimo così, il file `boolean.h` si presenterebbe in questo modo:

```
#define TRUE 1
#define FALSE 0
typedef int Bool;
```

Inserire le definizioni delle macro e dei tipi in un file header presenta alcuni chiari vantaggi. Per prima cosa, risparmiamo non dovendo copiare le definizioni in tutti i file sorgente che ne avessero bisogno. In secondo luogo il programma diventa più facile da modificare: cambiare la definizione di una macro o di un tipo richiede solo

la scrittura di un singolo file. Non dobbiamo modificare tutti i file dove la definizione viene utilizzata. Il terzo vantaggio è che non dobbiamo ~~occuparci~~ delle incoerenze causate da file diversi contenenti definizioni discordanti.

## Condividere i prototipi delle funzioni

Supponete che un file sorgente contenga una chiamata a una funzione `f` che è definita in un altro file, `foo.c`. Chiamare `f` senza prima dichiararla è rischioso. Senza un prototipo su cui basarsi, il compilatore viene forzato ad assumere che `f` restituiscia un `int` e che il numero di parametri combaci con il numero di argomenti presenti nella chiamata a `f`. Gli stessi argomenti vengono convertiti automaticamente in una specie di "formato standard" dalle promozioni di default degli argomenti [**promozione di default degli argomenti > 9.3**]. Le assunzioni fatte dal compilatore possono essere sbagliate ma questo non ha modo di controllarle poiché compila solamente un file alla volta. Se le assunzioni sono sbagliate, probabilmente il programma non funzionerà non ci saranno indizi sul perché (per questa ragione il C99 proibisce la chiamata a una funzione per la quale il compilatore non ha ancora incontrato una dichiarazione o una definizione).



Quando chiamate una funzione `f` che è stata definita in un altro file, assicuratevi che il compilatore abbia visto il prototipo di `f` prima della chiamata.

Il nostro primo impulso è quello di dichiarare `f` nel file nel quale viene chiamata. Questo risolve il problema ma può creare un incubo per la manutenzione. Supponete che la funzione venga chiamata in cinquanta file sorgente, come possiamo assicurare che i prototipi di `f` siano uguali in tutti i file? Come possiamo garantire che questi combacino con la definizione di `f` presente in `foo.c`? Se `f` dovesse essere modificata in un secondo momento, come potremmo individuare tutti i file nei quali è stata utilizzata?



La soluzione è ovvia: inserire il prototipo di `f` in un file header e poi includere questo file in tutti i luoghi nei quali `f` viene chiamata. Dato che `f` è stata definita in `foo.c`, chiamiamo l'header `foo.h`. Oltre a includere `foo.h` in tutti i file sorgente dove viene chiamata, dobbiamo includere questo file header anche in `foo.c` per permettere al compilatore di controllare che il prototipo di `f` presente in `foo.h` combaci con la definizione `foo.c`.



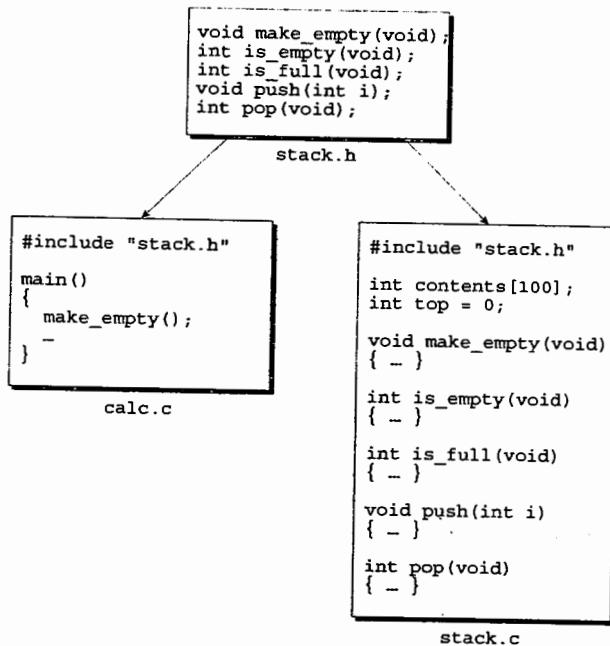
Includete sempre il file header che dichiara la funzione `f` all'interno del file sorgente che contiene la sua definizione. Non farlo può causare bachi difficili da trovare, dato che le chiamate alla `f` che si trovano in qualche altro punto del programma potrebbero non coincidere con la sua definizione.

Se `foo.c` contiene altre funzioni, la maggior parte di queste dovrebbe essere dichiarata nello stesso file header usato per `f`. Dopo tutto le altre funzioni in `foo.c` sono presumibilmente collegate a `f` e quindi un qualunque file contenente una chiamata a `f` probabilmente avrà bisogno di qualche altra funzione presente in `foo.c`. Le funzioni che sono pensate per essere utilizzate solo all'interno di `foo.c`, non dovrebbero essere dichiarate in un file header, farlo sarebbe fuorviante.

Per illustrare l'utilizzo dei prototipi delle funzioni nei file di header, ritorniamo alla calcolatrice RPN della Sezione 15.1. Il file stack.c conterrà le definizioni delle funzioni make\_empty, is\_empty, is\_full, push e pop. I seguenti prototipi per quelle funzioni dovrebbero essere inseriti nel file header stack.h:

```
void make_empty(void);
int is_empty(void);
int is_full(void);
void push(int i);
int pop(void);
```

(Per evitare di complicare l'esempio, le funzioni is\_empty e is\_full restituiranno dei valori int invece che dei valori Boolean.) Includeremo stack.h in calc.c per permettere che il compilatore possa controllare tutte le chiamate alle funzioni dello stack che compaiono nell'ultimo file. Dovremo includere anche stack.h in stack.c in modo che il compilatore possa verificare che i prototipi presenti in stack.h combacino con le definizioni presenti in stack.c. Le seguenti figure mostrano stack.h, stack.c e calc.c:



## Condividere la dichiarazione delle variabili

Le variabili esterne [variabili esterne > 10.2] possono essere condivise tra i file allo stesso modo con cui vengono condivise le funzioni. Per condividere una funzione mettiamo la sua definizione in un file sorgente, e poi inseriamo delle dichiarazioni

negli altri file che hanno bisogno di chiamare la funzione. La condivisione di una variabile esterna avviene praticamente allo stesso modo.

Fino a questo momento non abbiamo avuto bisogno di distinguere tra la dichiarazione di una variabile e la sua definizione. Per dichiarare una variabile i abbiamo scritto

```
int i; /* dichiara la variabile i e la definisce */
```

che non solo dichiara i come una variabile di tipo int ma allo stesso modo definisce anche i facendo sì che il compilatore riservi dello spazio per la variabile stessa. Per dichiarare la variabile i senza definirla dobbiamo mettere la keyword extern [keyword **extern > 18.2**] all'inizio della dichiarazione:

```
extern int i; /* dichiara i senza definirla */
```

la keyword extern informa il compilatore che i viene definita altrove nel programma (molto probabilmente in un altro file sorgente) e quindi che non c'è bisogno di allocare dello spazio per essa.

La keyword extern funziona con variabili di tutti i tipi. Quando la utilizziamo nella dichiarazione di un vettore possiamo omettere la lunghezza del vettore:

```
D&R
extern int a[];
```

Dato che il compilatore non alloca spazio per a, non ha alcun bisogno di conoscere la lunghezza del vettore.

Per condividere la variabile i tra più file sorgente, dobbiamo per prima cosa mettere la definizione di i in un file:

```
int i;
```

Se i ha bisogno di essere inizializzata, l'iniziatore deve andare qui. Quando il file viene compilato, il compilatore allocherà della memoria per i. Gli altri file conterranno delle dichiarazioni di i:

```
extern int i;
```

Dichiarando la variabile in ogni file diventa possibile accedere e/o modificare i all'interno di quei file. Tuttavia per effetto della parola extern, il compilatore non allocherà della memoria aggiuntiva per i ogni volta che uno di quei file viene compilato.

Quando una variabile viene condivisa tra più file, dobbiamo affrontare un problema simile a quello incontrato con le funzioni condivise: assicurarsi che tutte le dichiarazioni di una variabile coincidano con la definizione della stessa variabile.



Quando dichiarazioni della stessa variabile compaiono in file differenti, il compilatore non può controllare che le dichiarazioni combacino con la definizione della variabile. Per esempio, un file può contenere la definizione

```
int i;
```

mentre un altro file può contenere la dichiarazione

```
extern long i;
```

Un errore di questo tipo può causare un comportamento non predicable da parte del programma.

Per evitare inconsistenze, di solito le dichiarazioni di variabili condivise vengono inserite nei file header. Un file sorgente che avesse bisogno di accedere a una particolare variabile potrebbe includere l'header appropriato. Inoltre, ogni file header contenente la dichiarazione di una variabile viene incluso nel file sorgente che contiene la definizione di quest'ultima permettendo così al compilatore di controllare che non vi siano discrepanze.

Sebbene la condivisione delle variabili sia una vecchia pratica nel mondo del C, presenta seri svantaggi. Nella Sezione 19.2 vedremo quali sono i problemi che questa pratica comporta e impareremo come progettare programmi che non hanno bisogno di variabili condivise.

## include annidati

Anche un file header può contenere delle direttive #include. Sebbene questa pratica possa sembrare un po' strana, agli effetti pratici è piuttosto utile. Considerate il file stack.h contenente i seguenti prototipi:

```
int is_empty(void);
int is_full(void);
```

Dato che queste funzioni restituiscono solo 0 o 1, è una buona idea dichiarare il loro tipo restituito come Bool e non come int, dove Bool è un tipo che abbiamo definito precedentemente in questa sezione:

```
Bool is_empty(void);
Bool is_full(void);
```

Naturalmente abbiamo bisogno di includere il file boolean.h all'intero di stack.h in modo che la definizione di Bool sia disponibile al momento della compilazione di stack.h (nel C99 includiamo <stdbool.h> al posto di boolean.h e dichiareremo come bool invece che Bool il tipo restituito dalle due funzioni).

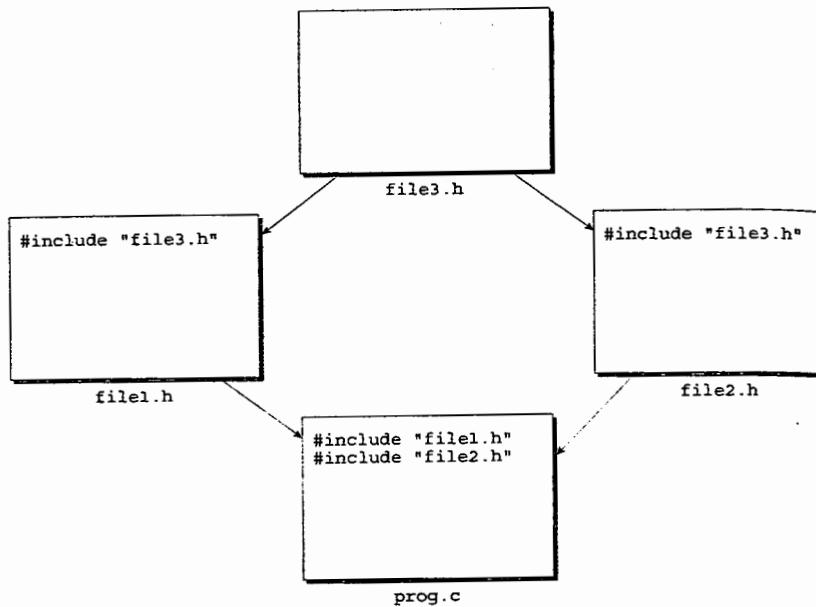
Tradizionalmente i programmati C evitano gli include annidati (le prime versioni del C non li permettevano affatto). Tuttavia la propensione avversa agli include annidati si è in parte affievolita grazie al fatto che questi rappresentano una pratica comune nel C++.

## Proteggere i file header

Se un file sorgente include lo stesso header due volte possono verificarsi degli errori di compilazione. Questo problema è comune quando i file header includono altri file header. Per esempio: supponete che file1.h includa file3.h, che file2.h includa file3.h e che prog.c includa sia file1.h che file2.h (si veda la figura alla pagina seguente). Quando prog.c viene compilato, file3.h viene compilato due volte.

Includere lo stesso header due volte non causa sempre degli errori di compilazione. Se il file contiene solo delle definizioni di macro, dei prototipi di funzioni, e/o delle dichiarazioni di variabili, allora non si verificherà alcun problema. Se però il file contiene la definizione di un tipo otterremo un errore di compilazione.

Per ragioni di sicurezza probabilmente è meglio proteggere i file di header dalle inclusioni multiple. In questo modo possiamo aggiungere successivamente delle definizioni di tipo senza il rischio di dimenticarci di proteggere il file.



In aggiunta potremmo risparmiare tempo durante lo sviluppo del programma evitando le ricompilazioni non necessarie dello stesso file header.

Per proteggere un file header richiuderemo il contenuto del file all'interno di una coppia `#ifndef-#endif`. Per esempio, il file `boolean.h` può essere protetto nel modo seguente:

```

#ifndef BOOLEAN_H
#define BOOLEAN_H

#define TRUE 1
#define FALSE 0
typedef int Bool;

#endif

```

Quando questo file viene incluso per la prima volta, la macro `BOOLEAN_H` non è definita e quindi il preprocessore permetterà alle righe comprese tra `#ifndef` ed `#endif` di rimanere. Se il file dovesse essere incluso una seconda volta, il preprocessore rimuove le righe comprese tra quelle due direttive.

Il nome della macro (`BOOLEAN_H`) non ha alcuna importanza, tuttavia fare in modo che assomigli al nome del file è un buon modo per evitare conflitti con altre macro. Dato che non possiamo chiamare la macro `BOOLEAN.H` (gli identificatori non possono contenere il punto), un nome come `BOOLEAN_H` è una buona alternativa.

## Direttive #error nei file header

Le direttive `#error` [direttive #error > 14.5] vengono inserite spesso nei file header per controllare delle condizioni sotto le quali il file header non dovrebbe essere incluso. Per esempio, supponete che un file header utilizzzi una funzionalità che non esisteva prima dello standard C89. Per prevenire l'utilizzo del file da parte di un compilatore non standard, l'header potrebbe contenere un direttiva `#ifndef` per controllare l'esistenza della macro `_STDC_` [macro \_STDC\_ > 14.3]:

```
#ifndef _STDC_
#error This header requires a Standard C compiler
#endif
```

## 15.3 Suddividere un programma su più file

Utilizziamo quanto sappiamo sui file header e sui file sorgente per sviluppare una semplice tecnica per dividere il programma su più file. Ci concentreremo sulle funzioni, tuttavia gli stessi principi possono essere applicati allo stesso modo alle variabili esterne. Assumeremo che il programma sia stato già progettato, ovvero dovremo decidere di quali funzioni del programma avremo bisogno e di come suddividerle in gruppi affini secondo una certa logica (discuteremo della progettazione di un programma nel Capitolo 19).

Ecco come procederemo: ogni insieme di funzioni verrà inserito in un file sorgente separato (utilizzeremo il nome `foo.c` per uno di questi file). In aggiunta creeremo un file header con lo stesso nome del file sorgente, ma con estensione `.h` (`foo.h` nel nostro caso). All'interno di `foo.h` inseriremo i prototipi delle funzioni definite in `foo.c`. (Le funzioni che sono progettate per essere utilizzate solamente all'interno di `foo.c` non hanno bisogno, e non devono, essere dichiarate in `foo.h`. La funzione `read_char` del nostro prossimo programma ne è un esempio.) Includeremo `foo.h` in ogni file sorgente che abbia bisogno di invocare una funzione definita in `foo.c`. Inoltre includeremo `foo.h` all'interno di `foo.c` in modo che il compilatore possa controllare che i prototipi presenti nel file header siano coerenti con le definizioni presenti nel file sorgente.

La funzione `main` andrà in un file il cui nome combacerà con quello del programma. Se vogliamo che un programma sia conosciuto come `bar`, allora la funzione `main` dovrà essere contenuta nel file `bar.c`. È possibile che, oltre al `main`, in quel file siano presenti anche altre funzioni che non vengono chiamate da altri file appartenenti al programma.

## Formattare del testo

Per illustrare la tecnica che abbiamo appena discusso, la applicheremo a un piccolo programma di formattazione del testo chiamato `justify`. Come input di esempio per il nostro programma utilizzeremo un file chiamato `quote` contenente le seguenti (e mal formattate) citazioni dal brano di Dennis M. Ritchie "The development of the C programming language" (in *History of Programming Language II*, a cura di T.J. Bergin Jr. e R.G. Gibson Jr., Addison-Wesley, 1996, pagg. 671-687):

C is quirky, flawed, and an enormous success. Although accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.

-- Dennis M. Ritchie

Per eseguire il programma dalla riga di comando di UNIX o Windows immettiamo il seguente comando:

`justify <quote`

Il simbolo < informa il sistema operativo che `justify` dovrà leggere dal file quote invece di accettare dell'input da tastiera. Questa caratteristica, supportata da UNIX, Windows e altri sistemi operativi, è chiamata reindirizzamento dell'input (*input redirection*) [**reindirizzamento dell'input > 22.1**]. Quando al programma `justify` viene fornito il file quote come input, produce il seguente output:

C is quirky, flawed, and an enormous success. Although accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments. -- Dennis M. Ritchie

L'output di `justify` comparirà sullo schermo, tuttavia è possibile salvarlo in un file utilizzando il reindirizzamento dell'output (*output redirection*) [**reindirizzamento dell'output > 22.1**]:

`justify <quote >newquote`

L'output di `justify` comparirà all'interno del file `newquote`.

In generale l'output di `justify` dovrà essere uguale al suo input, ma gli spazi aggiuntivi e le righe vuote verranno cancellati, mentre le righe normali verranno riempite e giustificate. "Riempire" una riga significa aggiungervi delle parole fino a quando la riga non fuoriesce dai suoi limiti. "Giustificare" una riga significa immettere degli spazi aggiuntivi tra le parole in modo che ogni riga abbia esattamente la stessa lunghezza (60 caratteri). La "giustificazione" deve essere fatta in modo che lo spazio tra le parole presenti in una riga sia uguale (o il più simile possibile). L'ultima riga dell'output non verrà giustificata.

Assumeremo che nessuna parola sia più lunga di 20 caratteri (un segno di interruzione viene considerato parte della parola alla quale è adiacente). Questo è un po' restrittivo naturalmente, ma una volta che il programma sarà stato scritto e verrà eseguito il debug, potremo facilmente aumentare il limite al punto che praticamente questo non verrà mai superato. Se il programma incontra una parola più lunga deve ignorare tutti i caratteri successivi ai primi 20 rimpiazzandoli con un singolo asterisco. Per esempio, la parola

antidisestablishmentarianism

verrebbe stampat \come

antidisestablishment\*

Adesso che sapete quello che il programma deve fare, è tempo di pensare alla sua progettazione. Inizieremo osservando che il programma non può scrivere le parole una alla volta come quando vengono lette. Dovrà invece memorizzarle in un "buffer di riga" fino a quando ce ne saranno a sufficienza per riempire una riga. Dopo un'ulteriore riflessione possiamo decidere che il cuore del programma sia un ciclo di questo tipo:

```
for (;;) {
 leggi parola;
 if (non si può leggere parola) {
 scrivi contenuto buffer di riga senza giustificazione;
 termina il programma;
 }
 if (la parola non entra nel buffer di riga) {
 scrivi contenuto buffer di riga con giustificazione;
 pulisci buffer di riga;
 }
 aggiungi parola nel buffer di riga;
}
```

Poiché sono necessarie funzioni che gestiscano le parole e funzioni che gestiscano il buffer di riga, divideremo il programma in tre file sorgente. Metteremo tutte le funzioni relative alle parole in un file (*word.c*) e tutte le funzioni relative al buffer di riga in un altro file (*line.c*). Un terzo file (*justify.c*) conterrà la funzione *main*. In aggiunta a questi file avremo bisogno di due file header: *word.h* e *line.h*. Il file *word.h* conterrà i prototipi per le funzioni presenti in *word.c*, mentre *line.h* giocherà un ruolo simile per *line.c*.

Esaminando il ciclo principale vediamo che la sola funzione relativa alle parole di cui abbiamo bisogno è *read\_word* (se *read\_word* non può leggere una parola perché ha raggiunto la fine del file di input, dovremo segnalarlo nel ciclo del *main* facendo finta di aver letto una parola "vuota"). Conseguentemente il file *word.h* è piuttosto piccolo:

```
word.h
#ifndef WORD_H
#define WORD_H

* read_word: legge la successiva parola dall'input e la
* memorizza. Fa diventare la parola una
* stringa vuota se nessuna parola può essere
* letta a causa della fine del file.
* Tronca la parola se la sua lunghezza eccede
* len.

void read_word(char *word, int len);
```

```
#endif
```

Osservate come la macro WORD\_H protegga word.h dall'essere incluso più di una volta. Sebbene word.h non ne abbia davvero bisogno, è una buona pratica proteggere tutti i file header in questo modo.

Il file line.h non sarà breve quanto word.h. Il nostro schema per ciclo del main rivela la necessità di funzioni che eseguano le seguenti operazioni:

- scrivere i contenuti del buffer di riga senza giustificazione;
- determinare quanti caratteri sono rimasti nel buffer di riga;
- scrivere i contenuti del buffer di riga con giustificazione;
- pulire il buffer di riga;
- aggiungere una parola nel buffer di riga.

Chiameremo queste funzioni flush\_line, space\_remaining, write\_line, clear\_line, add\_word. Ecco come si presenterà il file line.h:

```
line.h #ifndef LINE_H
#define LINE_H

/*****************
 * clear_line: Pulisce la riga corrente.
 *****************/
void clear_line(void);

/*****************
 * add_word: Aggiunge una parola alla fine della riga
 * corrente. Se non è la prima parola della
 * riga mette uno spazio prima della parola.
 *****************/
void add_word(const char *word);

/*****************
 * space_remaining: Restituisce il numero dei caratteri
 * rimanenti nella riga corrente.
 *****************/
int space_remaining(void);

/*****************
 * write_line: Scrive la riga corrente giustificandola.
 *****************/
void write_line(void);

/*****************
 * flush_line: Scrive la riga corrente senza
 * giustificazione. Se la riga e' vuota,
 * non fa nulla.
 *****************/
void flush_line(void);
#endif
```

Prima di scrivere i file word.c e line.c, possiamo utilizzare le funzioni dichiarate in word.h e line.h per scrivere il programma principale justify.c. Scrivere questo file è più che altro una questione di tradurre in C il nostro progetto originale per il ciclo.

```
justify.c /* Formatta un file di testo */

#include <string.h>
#include "line.h"
#include "word.h"

#define MAX_WORD_LEN 20

int main(void)
{
 char word[MAX_WORD_LEN+2];
 int word_len;

 clear_line();
 for (;;) {
 read_word(word, MAX_WORD_LEN+1);
 word_len = strlen(word);
 if (word_len == 0) {
 flush_line();
 return 0;
 }
 if (word_len > MAX_WORD_LEN)
 word[MAX_WORD_LEN] = '*';
 if (word_len + 1 > space_remaining()) {
 write_line();
 clear_line();
 }
 add_word(word);
 }
}
```

Includere sia line.h che word.h fornisce al compilatore l'accesso ai prototipi delle funzioni presenti in entrambi i file nel momento in cui compila justify.c.

La funzione main utilizza un trucco per gestire le parole che eccedono i 20 caratteri. Quando chiama read\_word, il main dice alla funzione di troncare tutte le parole che eccedono 21 caratteri. Dopo il termine della funzione read\_word, il main controlla se word contiene una stringa che è lunga 20 caratteri. Se è così, la parola che è stata letta deve essere lunga almeno 21 caratteri (prima del troncamento) e così il main sostituisce il ventunesimo carattere della parola con un asterisco.

Ora è il momento di scrivere word.c. Sebbene il file header word.h contenga il prototipo di una sola funzione (read\_word), se ne abbiamo bisogno possiamo inserire funzioni aggiuntive in word.c. read\_word è più facile da scrivere se aggiungiamo una piccola funzione di "aiuto": read\_char. A read\_char assegneremo il compito di leggere un singolo carattere. Se il carattere letto è un new-line o una tabulazione, questo viene convertito in uno spazio. Facendo sì che read\_word chiami read\_char invece che getchar viene risolto il problema di gestire come spazi i caratteri new-line e le tabulazioni.

Ecco il file word.c:

```
word.c #include <stdio.h>
#include "word.h"

int read_char(void)
{
 int ch = getchar();

 if (ch == '\n' || ch == '\t')
 return ' ';
 return ch;
}

void read_word(char *word, int len)
{
 int ch, pos = 0;

 while ((ch = read_char()) == ' ')
 ;
 while (ch != ' ' && ch != EOF) {
 if (pos < len)
 word[pos++] = ch;
 ch = read_char();
 }
 word[pos] = '\0';
}
```

Prima di iniziare la discussione sulla funzione `read_word`, spendiamo qualche parola sull'utilizzo di `getchar` nella funzione `read_char`. Per prima cosa, `getchar` restituisce un valore `int` invece di un valore `char` e questo è il motivo per cui il tipo restituito dalla funzione `read_char` è `int`. Inoltre anche `getchar` restituisce il valore `EOF` [macro `EOF` **22.4**] quando non è in grado di continuare la lettura (di solito perché ha raggiunto la fine del file di input).

La funzione `read_word` consiste di due cicli. Il primo ciclo salta gli spazi fermandosi al primo carattere non bianco (`EOF` non è bianco e quindi il ciclo si ferma se incontra la fine del file.) Il secondo ciclo legge i caratteri fino a quando non incontra uno spazio o `EOF`. Il corpo del ciclo salva i caratteri nella variabile `word` fino a che non viene raggiunto il limite `len`. Dopo, il ciclo continua leggendo i caratteri ma non salvandoli. L'istruzione finale presente in `read_word` termina la parola con il carattere null facendola diventare una stringa. Se `read_word` incontra `EOF` prima di trovare un carattere non bianco, allora al termine del ciclo la variabile `pos` avrà il valore 0 e così `word` corrisponderà a una stringa vuota.

L'unico file rimasto è `line.c`, che fornisce le definizioni delle funzioni dichiarate nel file `line.h`. Il file `line.c` avrà bisogno anche di alcune variabili per tenere traccia dello stato del buffer di linea. Una variabile (`line`) si occuperà di contenere i caratteri presenti nella riga corrente. Per la precisione, `line` è l'unica variabile di cui abbiamo bisogno. Tuttavia per velocità e per comodità, utilizzeremo altre due variabili: `line_len` (il numero di caratteri presenti nella riga corrente) e `num_words` (il numero di parole presenti nella riga corrente).

Ecco il file line.c:

```

line.c #include <stdio.h>
#include <string.h>
#include "line.h"
#define MAX_LINE_LEN 60

char line[MAX_LINE_LEN+1];
int line_len = 0;
int num_words = 0;

void clear_line(void)
{
 line[0] = '\0';
 line_len = 0;
 num_words = 0;
}

void add_word(const char *word)
{
 if (num_words > 0) {
 line[line_len] = ' ';
 line[line_len+1] = '\0';
 line_len++;
 }
 strcat(line, word);
 line_len += strlen(word);
 num_words++;
}

int space_remaining(void)
{
 return MAX_LINE_LEN - line_len;
}

void write_line(void)
{
 int extra_spaces, spaces_to_insert, i, j;

 extra_spaces = MAX_LINE_LEN - line_len;
 for (i = 0; i < line_len; i++) {
 if (line[i] != ' ')
 putchar(line[i]);
 else {
 spaces_to_insert = extra_spaces / (num_words - 1);
 for (j = 1; j <= spaces_to_insert + 1; j++)
 putchar(' ');
 extra_spaces -= spaces_to_insert;
 num_words--;
 }
 }
 putchar('\n');
}

```

```

 }

void flush_line(void)
{
 if (line_len > 0)
 puts(line);
}

```

La maggior parte delle funzioni presenti in `line.c` sono facili da scrivere. L'unica complessa è la `write_line` che scrive una riga giustificandola. Questa funzione scrive i caratteri presenti in `line` uno a uno fermandosi agli spazi compresi tra ogni coppia di parole per scriverne di addizionali, se necessario. Il numero di spazi addizionali viene memorizzato in `spaces_to_insert`, che ha il valore `extra_spaces / (num_words - 1)`, dove `extra_spaces` inizialmente è la differenza tra la massima lunghezza della riga e la lunghezza corrente della riga. Dato che `extra_spaces` e `num_words` cambiano dopo la stampa di ogni parola, la variabile `spaces_to_insert` cambia a sua volta. Se inizialmente `extra_spaces` è uguale a 10 e `num_words` è uguale a 5, allora la prima parola sarà seguita da 2 spazi addizionali, la seconda da 2 spazi, la terza da 3 e la quarta da 3.

## 15.4 Build di un programma costituito da più file

Nella Sezione 2.1 abbiamo esaminato il processo di compilazione e linking di un programma contenuto in un unico file. Espandiamo ora quella disamina per trattare il caso di un programma costituito da più file. Fare il build di un grande programma richiede l'esecuzione degli stessi passi base visti per i programmi su singolo file.

- **Compilazione.** Ogni file sorgente presente nel programma deve essere compilato separatamente. (I file header non necessitano di essere compilati. Il contenuto di un file header viene compilato automaticamente al momento della compilazione di un file sorgente che ne facesse l'inclusione.) Per ogni file sorgente, il compilatore genera un file contenente del codice oggetto. Questi file, conosciuti come file oggetto, hanno estensione `.o` in UNIX ed estensione `.obj` in Windows;
- **Linking.** Il linker combina i file oggetto creati nel passo precedente (insieme con il codice delle funzioni di libreria) al fine di produrre un file eseguibile. Tra gli altri compiti, il linker è responsabile della risoluzione dei riferimenti esterni lasciati dal compilatore (un riferimento esterno si verifica quando una funzione presente in un file invoca una funzione definita in un altro file oppure accede a una variabile definita in un altro file).

La maggior parte dei compilatori ci permette di effettuare il building di un programma in un singolo passo. Per esempio, per fare il building del programma `justify` della Sezione 15.3 con il compilatore GCC utilizzeremo il seguente comando:

```
gcc -o justify justify.c line.c word.c
```

Questi tre file sorgente vengono prima compilati in codice oggetto. I file oggetto vengono poi passati automaticamente al linker che li unisce per formare un singolo file. L'opzione `-o` specifica che vogliamo un file eseguibile chiamato `justify`.

## Makefile

Mettere i nomi di tutti i file sorgente sulla riga di comando diventa presto tedioso. Peggio ancora: se ricompiliamo tutti i file sorgente, e non solo quelli effettivamente modificati, perdiamo un sacco di tempo quando rifacciamo il building di un programma.

Per facilitare il building di grandi programmi, dall'ambiente UNIX ha avuto origine il concetto di makefile: un file contenente tutte le informazioni necessarie per fare il building di un programma. Un makefile non elenca solamente i file che fanno parte del programma ma descrive anche le dipendenze tra i file. Supponete che il file `foo.c` includa il file `bar.h`. In tal caso diciamo che `foo.c` "dipende" da `bar.h`, questo perché una modifica a `bar.h` richiederebbe di ricompilare `foo.c`.

Ecco un makefile UNIX per il programma `justify`. Il makefile usa GCC per la compilazione e il linking:

```
justify: justify.o word.o line.o
 gcc -o justify.o word.o line.o

justify.o: justify.c word.h line.h
 gcc -c justify.c

word.o: word.c word.h
 gcc -c word.c

line.o: line.c line.h
 gcc -c line.c
```

Ci sono quattro gruppi di righe, ogni gruppo viene chiamato regola. La prima riga di ogni regola definisce un file target seguito dai file dai quali dipende. La seconda riga è un comando che deve essere eseguito se deve essere rifatto il build del target a causa di una modifica a una delle sue dipendenze. Concentriamoci sulle prime due regole dato che le ultime due sono simili.

Nella prima regola il target è `justify` (il file eseguibile):

```
justify: justify.o word.o line.o
 gcc -o justify.o word.o line.o
```

La prima riga dice che `justify` dipende dai file `justify.o`, `word.o` e `line.o`. Se uno qualsiasi di questi file viene modificato dopo l'ultimo build del programma, allora il building di `justify` deve essere rieseguito. Il comando presente nella riga seguente indica come deve essere eseguito il building, ovvero usando il comando `gcc` per fare il linking dei tre file oggetto.

Nella seconda regola il target è `justify.o`:

```
justify.o: justify.c word.h line.h
 gcc -c justify.c
```

La prima riga indica che si debba rifare il building di `justify.o` nel caso ci fosse una modifica a `justify.c`, `word.h` o `line.h`. (La ragione per menzionare `word.h` è che `justify.c` include entrambi questi file e quindi risente di un eventuale modifica a uno di questi.) La riga successiva mostra come aggiornare `justify.o` (ricompilando `justify.c`).

L'opzione `-c` dice al compilatore di compilare `justify.c` in un file oggetto senza cercare di effettuare il linking.

**D&R** Una volta creato il makefile per un programma possiamo utilizzare l'utility `make` per fare il building del programma stesso (o per rifarlo). L'utility `make` può determinare quali file non sono aggiornati, controllando l'ora e la data associate a ogni file appartenente al programma.

Se volete provare `make` ecco alcuni dettagli che avete bisogno di conoscere:

- ogni comando presente nel makefile deve essere preceduto da un carattere tab non da una serie di spazi (nel nostro esempio i comandi sembrano indentati con otto spazi ma in effetti è un singolo carattere tab);
- un makefile viene normalmente contenuto in un file chiamato `Makefile` o `makefile`. Quando viene utilizzata, l'utility `make` controlla automaticamente l'esistenza di un file con uno di questi nomi all'interno della cartella corrente;
- per invocare `make` utilizzate il comando

`make target`

dove `target` è uno dei target elencati all'interno del makefile. Per fare il building dell'eseguibile `justify` utilizzando il nostro makefile, dovremo utilizzare il comando

`make justify`

- se non viene specificato alcun target al momento dell'invocazione di `make`, allo stesso tempo effettuerà il building del target della prima regola. Per esempio, il comando

`make`

effettuerà il building dell'eseguibile di `justify` dato che questo è proprio il primo target del nostro makefile. A eccezione della prima regola che gode di queste speciali proprietà, l'ordine delle altre regole presenti in un makefile può essere del tutto arbitrario.

L'utility `make` è tanto complessa che interi libri sono stati scritti al riguardo, per questo motivo non ci addentreremo ulteriormente nelle sue caratteristiche e potenzialità. Diremo solamente che di solito i makefile reali non sono così semplici come quello del nostro esempio. Ci sono diverse tecniche che riducono la ridondanza presente nei makefile e rendono più agevole la loro modifica, tuttavia ne riducono allo stesso tempo la leggibilità.

Non tutti utilizzano i makefile. Sono piuttosto diffusi anche altri strumenti per la manutenzione del software, inclusi i "file di progetto" (*project files*) supportati da alcuni ambienti di sviluppo integrati.

## Errori durante il linking

Alcuni errori non rilevabili durante la compilazione verranno trovati durante la fase di linking. In particolare, se la definizione di una funzione o una variabile è assente, il linker non sarà in grado di risolvere il suo riferimento esterno causando un messaggio di errore del tipo *undefined symbol* o *undefined reference*.

Gli errori rilevati dal linker di solito sono facili da correggere. Ecco alcune delle cause più comuni.

- **Errori di scrittura.** Se il nome di una variabile o di una funzione non viene digitato correttamente, il linker lo indicherà come mancante. Per esempio, se la funzione `read_char` fosse definita ma venisse invocata come `read_cahr`, il linker segnalerebbe la mancanza della funzione `read_char`.
- **File mancanti.** Se il linker non è in grado di trovare le funzioni appartenenti al file `foo.c`, potrebbe non sapere nulla di tale file. Controllate il makefile o il file di progetto per assicurarvi che anche `foo.c` sia elencato al suo interno.
- **Librerie mancanti.** Il linker potrebbe non essere in grado di trovare tutte le librerie di funzioni utilizzate all'interno del programma. Un esempio classico si verifica con programmi UNIX che utilizzano l'header `<math.h>`. La semplice inclusione dell'header nel programma potrebbe non essere sufficiente, infatti molte versioni di UNIX richiedono che al momento del linking del programma venga specificata l'opzione `-lm`. Questa opzione fa sì che il linker ricerchi un file di sistema contenente la versione compilata delle funzioni `<math.h>`. Non utilizzare questa opzione può causare la visualizzazione di un messaggio di "undefined reference" durante la fase di linking.

## Rieseguire il build di un programma

Durante lo sviluppo di un programma è rara la necessità di compilare tutti i suoi file. La maggior parte delle volte controlleremo il programma, lo modificheremo e rifaremo il building. Per poter risparmiare del tempo, il nuovo processo di building dovrebbe ricompilare solo i file che potrebbero essere interessati dalle ultime modifiche.

Assumete di aver progettato un programma nel modo indicato nella Sezione 15.3, ovvero con un file header per ogni file sorgente. Per vedere quanti file debbano essere ricompilati dopo una modifica dobbiamo considerare due possibilità.

La prima possibilità è che la modifica interessi solo un singolo file sorgente. In questo caso solamente quel file deve essere ricompilato (naturalmente dopo la ricompilazione si deve rifare il linking dell'intero programma). Considerate il programma `justify`. Supponete di voler comprimere la funzione `read_char` presente nel file `word.c` (le modifiche sono segnate in grassetto):

```
int read_char(void)
{
 int ch = getchar();

 return (ch == '\n' || ch == '\t') ? ' ' : ch;
}
```

Questa modifica non interessa `word.h` e quindi abbiamo bisogno solamente di ricompilare `word.c` e rieseguire il linking del programma.

La seconda possibilità è che la modifica interessi un file header. In questo caso dovremmo ricompilare tutti i file che includono il file header in questione, visto che potrebbero essere interessati dalla modifica (alcuni potrebbero non esserlo, ma è meglio essere prudenti).

A titolo di esempio consideriamo la funzione `read_word` del programma `justify.c`. Osservate che il `main` invoca la `strlen` immediatamente dopo aver invocato la `read_word` in modo da determinare la lunghezza della parola che è appena stata letta. Dato che la `read_word` conosce già la lunghezza della parola (la variabile `pos` della funzione `read_word` tiene traccia della lunghezza), sembra sciocco utilizzare la funzione `strlen`. Modificare `read_word` per restituire la lunghezza della parola letta è facile. Per prima cosa modifichiamo il prototipo di `read_word` presente in `word.h`:

```

* read_word: legge la prossima parola dall'input e la
* memorizza. Fa diventare la parola una
* stringa vuota se nessuna parola può essere
* letta a causa della fine del file.
* Tronca la parola se la sua lunghezza eccede
* len. Restituisce il numero dei caratteri
* memorizzati

int read_word(char *word, int len);
```

Naturalmente dobbiamo ricordarci di modificare i commenti che accompagnano il prototipo. Successivamente modifichiamo la definizione di `read_word` presente in `word.c`:

```
int read_word(char *word, int len)
{
 int ch, pos = 0;

 while ((ch = read_char()) == ' ')
 ;
 while (ch != ' ' && ch != EOF) {
 if (pos < len)
 word[pos++] = ch;
 ch = read_char();
 }
 word[pos] = '\0';
 return pos;
}
```

Infine modifichiamo `justify.c` rimuovendo l'include a `<string.h>` e modificando la funzione `main` in questo modo:

```
int main(void)
{
 char word[MAX_WORD_LEN+2];
 int word_len;

 clear_line();
 for (;;) {
 word_len = read_word(word, MAX_WORD_LEN+1);
 if (word_len == 0) {
 flush_line();
 return 0;
 }
 }
}
```

```

 if (word_len > MAX_WORD_LEN)
 word[MAX_WORD_LEN] = '*';
 if (word_len + 1 > space_remaining()) {
 write_line();
 clear_line();
 }
 add_word(word);
}
}

```

Una volta apportate queste modifiche, rifacciamo il building del programma ricompilando `word.c` e `justify.c` oltre a rieseguire il linking. Non c'è nessun bisogno di ricompilare `line.c` che non include `word.h` e quindi non verrà toccata dalle modifiche a quest'ultimo. Con il compilatore GCC possiamo utilizzare il seguente comando per rifare il building del programma:

```
gcc -o justify justify.c word.c line.o
```

Fate caso al riferimento al file `line.o` invece che al file `line.c`.

Uno dei vantaggi nell'utilizzo dei makefile è quello che ogni nuova fase di building viene gestita automaticamente. L'utility `make`, esaminando la data di ogni file, può determinare quali tra questi hanno subito modifiche dopo l'ultima fase di building. L'utility ricompila questi file assieme a tutti i file da essi dipendenti (sia direttamente che indirettamente). Per esempio, se effettuiamo le modifiche indicate nei file `word.h`, `word.c` e `justify.c` e poi eseguiamo il building del programma `justify`, allora l'utility `make` eseguirà le seguenti azioni:

1. effettua il building del file `justify.o` compilando `justify.c` (perché `justify.c` e `word.h` sono stati modificati);
2. effettua il building di `word.o` compilando `word.c` (perché `word.c` e `word.h` sono stati modificati);
3. effettua il building di `justify` facendo il linking di `justify.o`, `word.o` e `line.o` (perché `justify.o` e `word.o` sono stati modificati).

## Definire la macro al di fuori di un programma

Di solito i compilatori C forniscono dei metodi per specificare il valore di una macro nel momento della compilazione di un programma. Questa possibilità facilita la modifica del valore di una macro senza modificare nessun file del programma. Ciò è particolarmente utile quando il building dei programmi viene automatizzato utilizzando i makefile.

La maggior parte dei compilatori (GCC incluso) supporta l'opzione `-D` che permette di specificare il valore di una macro dalla riga di comando:

```
gcc -DDEBUG=1 foo.c
```

In questo esempio la macro `DEBUG` è definita in modo da assumere il valore 1 nel programma `foo.c`, proprio come se la riga

```
#define DEBUG 1
```

si trovasse all'inizio di foo.c. Se l'opzione -D definisce una macro senza specificarne il valore, questo viene assunto uguale a 1.

Molti compilatori supportano anche l'opzione -U che "annulla" la definizione di una macro come se venisse utilizzata la direttiva #undef. Possiamo utilizzare -U per annullare la definizione di una macro predefinita [macro predefinite > 14.3] o una che è stata definita precedentemente nella riga di comando con l'opzione -D.

## Domande & Risposte

**D:** Non ha fornito alcun esempio dell'uso della direttiva #include per l'inclusione di un file sorgente. Cosa succederebbe se lo facessimo?

**R:** Questa non sarebbe una buona pratica, sebbene non sia proibita. Qui c'è un esempio del tipo di problemi ai quali si andrebbe incontro. Supponete che foo.c definisca una funzione f della quale abbiamo bisogno nei file bar.c e baz.c. Per questo motivo nei due file mettiamo la direttiva

```
#include "foo.c"
```

Tutti i file verrebbero compilati correttamente. Il problema si verificherebbe più tardi quando il linker scopre due copie del codice oggetto per la funzione f. Naturalmente potremmo risolvere il problema includendo foo.c solo in bar.c e non in baz.c. Per evitare problemi è meglio utilizzare la direttiva #include solo con i file header e non con i file sorgente.

**D:** Quali sono le esatte regole di ricerca della direttiva #include? [p. 363]

**R:** Questo dipende dal vostro compilatore. Lo standard C si mantiene deliberatamente vago nella descrizione della direttiva #include. Se il nome del file è racchiuso tra parentesi acute, il preprocessore cerca, come dice lo standard, in una "sequenza di luoghi dipendenti dall'implementazione". Se il nome del file è racchiuso tra doppi apici, il file "viene cercato in un modo dipendente dall'implementazione" e, se non trovato, viene cercato come se fosse racchiuso tra parentesi acute. La ragione è semplice: non tutti i sistemi operativi possiedono un file system gerarchico (ad albero).

A rendere le cose ancora più interessanti è il fatto che lo standard non richiede che i nomi racchiusi tra parentesi acute siano dei nomi di file. In questo modo viene lasciata aperta la possibilità che le direttive #include che utilizzano le parentesi acute vengano gestite interamente all'interno del compilatore.

**D:** Non capiamo perché ogni file sorgente abbia bisogno di un suo file header. Perché non viene utilizzato un unico file header contenente tutte la definizioni di macro, le definizioni di tipo e i prototipi delle funzioni? Includendo questo header ogni file sorgente avrebbe accesso a tutte le informazioni necessarie. [p. 366]

**R:** L'approccio dell'unico grande file header funziona e un certo numero di programmatore lo utilizza. Possiede anche un vantaggio: avendo un unico file header ci sono meno file da gestire. Per i programmi più grandi però, gli svantaggi di questo approccio tendono a superare i vantaggi.

Utilizzando un singolo file header non viene fornita alcuna informazione utile a chi legge il programma. Con più file header il lettore può individuare velocemente

quali sono le altre parti di un programma che vengono utilizzate da un particolare file sorgente.

Questo non è tutto. Dato che ogni file sorgente dipende dal grande file header, modificarlo causerebbe la ricompilazione di tutti i file sorgente (uno svantaggio significativo nei grandi programmi). A peggiorare le cose si ha che il file header dovrà essere modificato spesso a causa del notevole quantitativo di informazioni in esso contenute.

**D: Il capitolo dice che un vettore condiviso dovrebbe essere dichiarato in questo modo:**

```
extern int a[];
```

**Dato che vettori e puntatori sono strettamente collegati, sarebbe ammисibile scrivere**

```
extern int *a;
```

**al posto della dichiarazione già vista? [p. 368]**

R: No. Quando utilizzati all'interno delle espressioni, i vettori "decadono" diventando puntatori (abbiamo notato questo comportamento quando il nome di un vettore viene utilizzato come un argomento in una chiamata a funzione). Nelle dichiarazioni delle variabili però, vettori e puntatori sono tipi distinti.

**D: Crea qualche problema includere in un file sorgente dei file header non necessari?**

R: No, a meno che il file header non contenga una dichiarazione o una definizione che va in conflitto con uno dei file sorgente. Altrimenti il peggio che può accadere è un piccolo incremento del tempo richiesto per compilare il file sorgente.

**D: Dobbiamo chiamare una funzione del file foo.c e per questo abbiamo incluso il file header corrispondente foo.h. Il programma è stato compilato correttamente, ma il linking non ha avuto successo. Perché?**

R: Nel C la compilazione e il linking sono due processi completamente separati. I file header esistono per fornire delle informazioni al compilatore e non al linker. Se volete chiamare una funzione presente in foo.c, allora dovete assicurarvi che foo.c venga compilato e che il linker sia a conoscenza del fatto che deve cercare il file oggetto foo.o per cercare la funzione. Di solito questo significa nominare il file foo.c nel makefile del programma o nel file di progetto.

**D: Se il nostro programma chiama una funzione presente in <stdio.h> questo significa che viene fatto il linking al programma di tutte le funzioni di <stdio.h>?**

R: No. Includere <stdio.h> (o ogni altro header) non ha effetti sul linking. Infatti la maggior parte dei linker effettuerà il linking delle sole funzioni effettivamente necessarie al vostro programma.

**D: Dove possiamo reperire l'utility make? [p. 380]**

R: make è un utility standard di UNIX. La versione GNU, conosciuta anche come GNU Make, viene inclusa nella maggior parte delle distribuzioni Linux. È anche direttamente disponibile presso la Free Software Foundation ([www.gnu.org/software/make/](http://www.gnu.org/software/make/)).

## Esercizi

- Sezione 15.1**
- La Sezione 15.1 ha elencato diversi vantaggi derivanti dalla suddivisione di un programma in più file sorgente.
    - Descrivete altri vantaggi.
    - Descrivete qualche svantaggio.
- Sezione 15.2**
- Quale dei seguenti non deve essere inserito in un file header? Perché no?
    - Prototipi di funzioni.
    - Definizioni di funzioni.
    - Definizioni di macro.
    - Definizioni di tipi.
  - Abbiamo visto che scrivere `#include <file>` invece di `#include "file"` può non funzionare se file è stato scritto da noi. Si verificherebbe qualche problema scrivendo `#include "file"` al posto di `#include <file>` se file fosse un header di sistema?
  - Assumete che `debug.h` sia un file header con i seguenti contenuti:

```
#ifdef DEBUG
#define PRINT_DEBUG(n) printf("Value of " #n ": %d\n", n)
#else
#define PRINT_DEBUG(n)
#endif
```

Il programma `testdebug.c` corrisponde al seguente file sorgente:

```
#include <stdio.h>

#define DEBUG
#include "debug.h"

int main(void)
{
 int i = 1, j = 2, k = 3;

#ifndef DEBUG
 printf("Output if DEBUG is defined:\n");
#else
 printf("Output if DEBUG is not defined:\n");
#endif

 PRINT_DEBUG(i);
 PRINT_DEBUG(j);
 PRINT_DEBUG(k);
 PRINT_DEBUG(i + j);
 PRINT_DEBUG(2 * i + j - k);

 return 0;
}
```

- (a) Qual è l'output del programma?
- (b) Qual è l'output del programma se dal file `testdebug.c` viene rimossa la direttiva `#define`?
- (c) Spiegate perché l'output del programma differisce tra le versioni delle domanda (a) e (b).
- (d) Al fine di ottenere l'effetto desiderato dalla macro `PRINT_DEBUG`, è necessario che la macro `DEBUG` sia definita prima di `debug.h`? Giustificate la vostra risposta.

**Sezione 15.4** 5. Supponete che un programma consista di tre file sorgente (`main.c`, `f1.c` ed `f2.c`) e di due file header (`f1.h` e `f2.h`). Scrivete un makefile per questo programma assumendo che il compilatore sia GCC e che il file eseguibile debba chiamarsi `demo`.

- W 6. Le domande seguenti si riferiscono al programma descritto nell'Esercizio 5.
  - (a) Quali file hanno bisogno di essere compilati quando il building del programma viene fatto per la prima volta?
  - (b) Se `f1.c` viene modificato dopo che il programma è stato compilato, quali file devono essere ricompilati?
  - (c) Se `f1.h` viene modificato dopo che il programma è stato compilato, quali file devono essere ricompilati?
  - (d) Se `f2.h` viene modificato dopo che il programma è stato compilato, quali file devono essere ricompilati?

## Progetti di programmazione

1. Il programma `justify` della Sezione 15.3 giustifica le righe inserendo degli spazi aggiuntivi tra le parole. Il modo nel quale la funzione `write_line` lavora attualmente fa sì che tra le parole vicine alla fine della riga ci siano degli spazi più ampi rispetto alle parole vicine all'inizio (per esempio, le parole vicine alla fine possono avere tre spazi tra di esse mentre quelle vicine all'inizio possono essere separate solamente da due spazi). Migliorate il programma in modo che `write_line` alterni tra l'inizio e la fine delle righe l'inserimento degli spazi più larghi.
2. Modificate il programma `justify` della Sezione 15.3 in modo che la funzione `read_word` (al posto del `main`) salvi il carattere `*` alla fine di una parola che è stata troncata.
3. Modificate il programma `qsort.c` della Sezione 9.6 in modo che le funzioni `quicksort` e `split` si trovino su un file separato chiamato `quicksort.c`. Create un file header chiamato `quicksort.h` che contenga i prototipi delle due funzioni e fate in modo che sia `qsort.c` che `quicksort.c` includano questo file.

4. Modificate il programma `remind.c` della Sezione 13.5 in modo che la funzione `read_line` si trovi in un file separato chiamato `readline.c`. Create un file header chiamato `readline.h` che contenga il prototipo della funzione e fate in modo che sia `remind.c` che `readline.c` includano questo file.
5. Modificate il Progetto di programmazione 6 del Capitolo 10 in modo che, come descritto nella Sezione 15.2, abbia due file separati `stack.h` e `stack.c`.

# 16 Strutture, unioni ed enumerazioni

Questo capitolo introduce tre nuovi tipi: strutture, unioni ed enumerazioni. Una struttura è una collezione di valori (numeri), anche di tipo diverso. Un'unione è simile a una struttura ma differisce da questa per il fatto che i suoi membri condividono la stessa area di memoria e, dunque, può salvare un membro per volta e non tutti i membri simultaneamente. Un'enumerazione è un tipo intero i cui valori hanno nomi scelti dal programmatore.

Di questi tre tipi le strutture sono di gran lunga il più importante e quindi vi dedicheremo gran parte del capitolo. La Sezione 16.1 mostra come dichiarare delle variabili struttura e come eseguire su di esse operazioni basilari. La Sezione 16.2 spiega come definire dei tipi struttura che, tra le altre cose, ci permettono di scrivere funzioni che accettano argomenti struttura o che restituiscono strutture. La Sezione 16.3 illustra come possano essere annidati vettori e strutture. Le ultime due sezioni sono dedicate alle unioni (Sezione 16.4) e alle enumerazioni (Sezione 16.5).

## 16.1 Variabili struttura

L'unica struttura dati che abbiamo incontrato finora è il vettore. I vettori presentano due importanti proprietà. La prima è che tutti gli elementi di un vettore sono dello stesso tipo; la seconda è che per selezionare un elemento di un vettore specifichiamo la sua posizione (sotto forma di indice intero).

Le proprietà di una **struttura** sono piuttosto differenti da quelle di un vettore. Gli elementi di una struttura (i suoi **membri**, nel gergo C) non devono essere dello stesso tipo. Inoltre i membri di una struttura hanno dei nomi. Per selezionare un particolare membro dobbiamo specificare il suo nome e non la sua posizione.

La maggior parte dei linguaggi di programmazione prevede le strutture: in alcuni linguaggi vengono chiamate **record** mentre i membri sono conosciuti come **campi** (*field*).

### Dichiarare variabili struttura

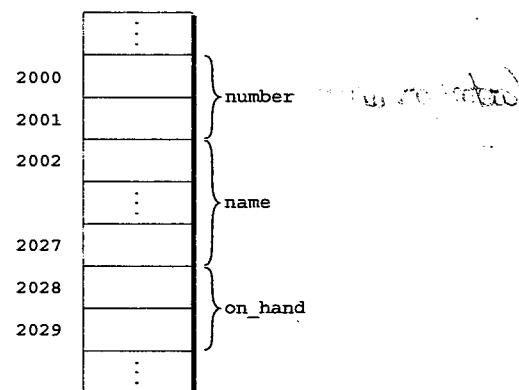
Quando dobbiamo memorizzare una collezione di dati concettualmente collegati, una struttura è la scelta più logica. Per esempio, supponete di dover tenere traccia dei

componenti presenti in un magazzino. Le informazioni che dobbiamo conservare per ogni oggetto devono includere: il numero del componente (un intero), il nome del componente (una stringa di caratteri) e il numero di componenti disponibili (un intero). Per creare delle variabili in grado di immagazzinare tutti e tre i tipi di dato, possiamo utilizzare una dichiarazione come la seguente:

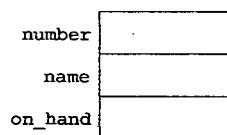
```
struct {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} part1, part2;
```

Ogni variabile struttura possiede tre membri: number (il numero del componente), name (il nome del componente) e on\_hand (la quantità disponibile). Osservate che questa dichiarazione ha lo stesso formato delle altre dichiarazioni di variabili viste in C. La notazione struct { ... } specifica un tipo, mentre part1 e part2 sono variabili di quel tipo.

I membri di una struttura vengono immagazzinati nella memoria nell'ordine con il quale sono stati dichiarati. Per mostrare come la variabile part1 si presenta in memoria, assumiamo che: (1) la variabile venga allocata all'indirizzo 2000, (2) gli interi occupino 4 byte, (3) NAME\_LEN possegga il valore 25 e (4) non ci siano spazi tra i membri della struttura. Con queste assunzioni part1 si presenta come segue:



Solitamente non è necessario disegnare le strutture con questo dettaglio. Normalmente le rappresenteremo in modo più astratto, come una serie di contenitori:



Qualche volta le raffigureremo orizzontalmente e non verticalmente:

|  |  |  |
|--|--|--|
|  |  |  |
|--|--|--|

number name on\_hand

Per esempio è un contenitore.

I valori dei membri verranno messi nei contenitori in un secondo momento, per ora li lasciamo vuoti.

Ogni struttura rappresenta un nuovo scope: ogni nome dichiarato all'interno di quello scope non andrà in conflitto con gli altri nomi del programma (nella terminologia C si dice che ogni struttura ha uno spazio dei nomi per i suoi membri). Per esempio, le seguenti dichiarazioni possono comparire all'interno dello stesso programma:

```
struct {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} part1, part2;

struct {
 char name[NAME_LEN+1];
 int number;
 char sex; (solo caratteri)
} employee1, employee2;
```

I membri number e name nelle strutture part1 e part2 non entrano in conflitto con i membri di number e name delle strutture employee1 ed employee2.

## Inizializzare variabili struttura

Come un vettore, anche una variabile struttura può essere inizializzata nello stesso momento in cui viene dichiarata. Per inizializzare una struttura dobbiamo preparare un elenco di valori che devono essere immagazzinati al suo interno e racchiudere questo elenco tra parentesi graffe:

```
struct {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} part1 = {528, "Disk drive", 10},
part2 = {914, "Printer cable", 5};
```

I valori dell'inizializzatore devono comparire nello stesso ordine dei membri della struttura. Nel nostro esempio il membro number della struttura part1 diventerà uguale a 528, il membro name a "Disk drive", e così via. A pagina seguente vediamo come si presenterà part1 dopo l'inizializzazione.

|         |            |
|---------|------------|
| number  | 528        |
| name    | Disk drive |
| on_hand | 10         |

Gli inizializzatori delle strutture seguono delle regole simili a quelle degli inizializzatori dei vettori. Le espressioni utilizzate all'interno di un inizializzatore di struttura devono essere costanti. Per esempio non possiamo utilizzare una variabile per inizializzare il membro `on_hand` di `part1` (come vedremo nella Sezione 18.5, questa restrizione è stata attenuata nel C99). Un inizializzatore può avere un numero di membri inferiore a quello della struttura che sta inizializzando. Così come succede con i vettori, tutti i membri che sono stati tralasciati avranno lo 0 come loro valore iniziale. In particolare, i byte tralasciati di un vettore di caratteri saranno uguali a zero, facendo sì che il vettore rappresenti una stringa vuota.

## Inizializzatori designati

Gli inizializzatori designati discussi nel contesto dei vettori all'interno della Sezione 8.1, possono essere utilizzati anche con le strutture. Considerate l'inizializzatore per `part1` visto nell'esempio precedente:

```
{528, "Disk drive", 10}
```

Un inizializzatore designato avrà un aspetto simile, ma ogni valore verrà etichettato con il nome del membro che inizializza:

```
{.number = 528, .name = "Disk drive", .on_hand = 10}
```

La combinazione formata dal punto e dal nome del membro viene chiamata **designatore** (i designatori per gli elementi di un vettore hanno un formato diverso).

Gli inizializzatori designati presentano diversi vantaggi. Per prima cosa sono più facili da leggere e da controllare perché il lettore può vedere chiaramente la corrispondenza tra i membri della struttura e i valori elencati nell'inizializzatore. Un altro vantaggio è dato dal fatto che i valori dell'inizializzatore non devono essere inseriti con lo stesso ordine con il quale i membri di una struttura sono elencati. Il nostro inizializzatore di esempio avrebbe potuto essere scritto in questo modo:

```
{.on_hand = 10, .name = "Disk drive", .number = 528}
```

Dato che l'ordine non ha importanza, il programmatore non deve ricordarsi dell'ordine nel quale i membri sono stati dichiarati originariamente. Inoltre, l'ordine dei membri può essere modificato in futuro senza incidere sui vari designatori inizializzati.

Non tutti i valori elencati in un inizializzatore designato devono essere prefissati da un designatore (come abbiamo visto nella Sezione 8.1 questo è vero anche per i vettori). Considerate il seguente esempio:

```
{.number = 528, "Disk drive", .on_hand = 10}
```

Il valore "Disk drive" non ha un designatore e quindi il compilatore assume che questo inizializzi il membro che segue `number` nella struttura. Tutti i membri per i quali l'inizializzatore non fornisce un valore vengono inizializzati al valore zero.

## Operazioni sulle strutture

Dato che l'operazione più comune sui vettori è l'indicizzazione (selezionare un elemento a partire dalla sua posizione), non deve sorprendere che l'operazione più comune su una struttura sia la selezione di uno dei suoi membri. I membri di una struttura sono accessibili a partire dal nome e non dalla posizione.

Per accedere a un membro all'interno di una struttura, prima scriviamo il nome della struttura, poi un punto e infine il nome del membro. Per esempio, le seguenti istruzioni visualizzeranno i valori dei membri di `part1`:

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

I membri di una struttura sono degli lvalue [value > 4.2] e quindi possono comparire sul lato sinistro di un assegnamento oppure come operandi in un'espressione di incremento o decremento:

```
part1.number = 258; /* modifica il numero di componenti di part1 */
part1.on_hand++; /* incrementa i componenti disponibili di part1 */
```

Il punto che utilizziamo per accedere al membro di una struttura è agli effetti pratici un operatore del C. Questo possiede lo stesso ordine di precedenza degli operatori suffisso `++` e `--` [tabella degli operatori > Appendice A]. Di conseguenza ha precedenza su quasi tutti gli altri operatori. Considerate l'esempio seguente:

```
scanf("%d", &part1.on_hand);
```

L'espressione `&part1.on_hand` contiene due operatori (`&` e `.`). L'operatore punto ha precedenza sull'operatore `&`, di conseguenza `&` calcola l'indirizzo di `part1.on_hand`.

L'altra importante operazione sulle strutture è l'assegnamento:

```
part2 = part1;
```

L'effetto di questa istruzione è quello di copiare `part1.number` in `part2.number`, `part1.name` in `part2.name` e così via.

Dato che i vettori non possono essere copiati utilizzando l'operatore `=`, sembra strano scoprire che le strutture lo possono fare. È anche più sorprendente se considerate che un vettore incorporato all'interno di una struttura viene copiato quando la struttura che lo contiene viene copiata. Alcuni programmati sfruttano questa proprietà per creare delle strutture fittizie contenenti vettori che verranno copiati in un secondo momento:

```
struct { int a[10]; } a1, a2;
```

```
a1 = a2; /* ammissibile, visto che a1 e a2 sono strutture */
```

L'operatore = può essere usato solo con strutture di tipo compatibile. Due strutture dichiarate allo stesso momento (come part1 e part2) sono compatibili. Come vedremo nella prossima sessione, le strutture dichiarate utilizzando lo stesso "tag di struttura" o lo stesso nome di tipo sono anch'esse compatibili.

Oltre agli assegnamenti il C non fornisce altre operazioni che operano sull'intera struttura. In particolare, non possiamo utilizzare gli operatori == e != per controllare se due strutture sono uguali o meno.

D&amp;R

## 16.2 Tipi struttura

La sezione precedente ha mostrato come dichiarare delle variabili struttura, ma non ha trattato un argomento importante: dare il nome ai tipi struttura. Supponete che un programma debba dichiarare diverse variabili struttura con membri identici. Se tutte le strutture possono essere dichiarate in una volta sola, allora non ci sono problemi. Se tuttavia dobbiamo dichiarare le variabili in punti diversi del programma allora tutto risulta più difficile. Se in un punto scriviamo

```
struct {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} part1;
```

e in un altro

```
struct {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} part2;
```

allora incontreremo dei problemi. Ripetere l'informazione della struttura renderà più grande il programma. Modificare il programma in un secondo momento sarebbe rischioso, dato che non possiamo garantire facilmente che le dichiarazioni rimangano consistenti.

Tuttavia questi non sono i problemi più gravi. Secondo le regole del C, part1 e part2 non hanno dei tipi compatibili. Come risultato si ha che part1 non può essere assegnato a part2 e viceversa. Inoltre, dato che non abbiamo un nome per il tipo di part1 o part2, non possiamo utilizzarli come argomenti in una chiamata di funzione.

D&amp;R

Per evitare queste difficoltà avremo bisogno di poter definire un nome che rappresenti il tipo della struttura e non una particolare variabile. Il C fornisce due modi per dare il nome alle strutture: possiamo sia dichiarare un "tag di struttura" o utilizzare typedef per definire un nome di tipo [definizioni di tipo > 7.5].

### Dichiarare il tag di struttura

Il tag di struttura è il nome utilizzato per identificare un particolare tipo di struttura. L'esempio seguente dichiara un tag di struttura chiamato part:

```
struct part { (valore fissa, da cancellare) }; (part)
 int number;
 char name[NAME_LEN+1];
 int on_hand;
};
```

Notate il punto e virgola che segue la parentesi graffa destra e che deve essere presente per terminare la dichiarazione.



Omettere accidentalmente il punto e virgola alla fine della dichiarazione di una struttura può causare degli errori sconcertanti. Considerate l'esempio seguente:

```
struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} *** SBAGLIATO: manca il punto e virgola ***/

f(void)
{
 return 0; /* errore rilevato in questa riga */
}
```

Il programmatore non ha specificato il tipo restituito dalla funzione f (uno stile di programmazione un po' trascurato). Dato che la precedente dichiarazione di struttura non era stata terminata correttamente, il compilatore assume che f restituisca un valore di tipo struct part. L'errore non verrà rilevato fino a quando il compilatore non raggiunge la prima istruzione return all'interno della funzione. Il risultato è un criptico messaggio di errore.

Una volta creato il tag part, possiamo utilizzarlo per dichiarare delle variabili:

```
struct part part1, part2;
```

Sfortunatamente non possiamo abbreviare questa dichiarazione eliminando la parola struct:

```
part part1, part2; *** SBAGLIATO ***/
```

part non è il nome di un tipo, di conseguenza senza la parola struct non ha alcun significato.

Poiché i tag non vengono riconosciuti, a meno che non siano preceduti dalla parola struct, non andranno in conflitto con gli altri nomi utilizzati in un programma. Sarebbe perfettamente ammissibile (sebbene genererebbe non poca confusione) che una variabile venisse chiamata part.

Tra l'altro la dichiarazione di un tag di struttura può essere combinata con la dichiarazione delle variabili struttura:

```
struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} part1, part2;
```

Nel codice appena visto abbiamo dichiarato un tag di struttura chiamato `part` (rendendo possibile l'utilizzo di `part` per una futura dichiarazione di altre variabili) e al contempo abbiamo dichiarato le variabili `part1` e `part2`.

Tutte le strutture dichiarate del tipo `struct part` sono compatibili tra loro:

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;
part2 = part1; /* ammissibile; sono dello stesso tipo */
```

## Definire un tipo struttura

Come alternativa alla dichiarazione di un tag struttura, possiamo utilizzare `typedef` per definire un vero nome di tipo. Possiamo per esempio definire un tipo chiamato `Part` nel modo seguente:

```
typedef struct {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} Part;
```

Osservate che il nome del tipo, `Part`, deve comparire alla fine e non dopo la parola `struct`.

Possiamo utilizzare `Part` allo stesso modo dei tipi nativi del linguaggio. Per esempio possiamo utilizzarlo per dichiarare delle variabili:

```
Part part1, part2;
```

Dato che `Part` è un nome `typedef`, la scrittura `struct Part` non ci è permessa. Tutte le variabili `Part` sono compatibili indipendentemente da dove queste siano state dichiarate.

Quando viene il momento di dare il nome a una struttura, di solito possiamo scegliere di dichiarare o un tag di struttura o di utilizzare `typedef`. Tuttavia, come vedremo più avanti, dichiarare un tag di struttura è obbligatorio quando la struttura viene utilizzata in una lista concatenata [liste concatenate > 17.5]. Nella maggior parte dei nostri esempi utilizzeremo tag di struttura piuttosto che nomi `typedef`.

## Strutture come argomenti e valori restituiti

Le funzioni possono utilizzare le strutture come argomenti e come valore restituito. Analizziamo due esempi. La nostra prima funzione stampa i membri della struttura `part` che le viene passata come argomento:

```
void print_part(struct part p)
{
 printf("Part number: %d\n", p.number);
```

```
 } printf("Part name: %s\n", p.name);
 printf("Quantity on hand: %d\n", p.on_hand);
}
```

Ecco come potrebbe essere invocata la funzione print\_part:

```
print_part(part1);
```

La nostra seconda funzione restituisce una struttura part che viene costruita a partire dagli argomenti:

```
struct part build_part(int number, const char *name,
 int on_hand)
{
 struct part p;
 p.number = number;
 strcpy(p.name, name);
 p.on_hand = on_hand;
 return p;
}
```

Osservate che ai parametri di build\_part è ammesso possedere nomi che corrispondono con i membri della struttura part dato che essa possiede il proprio spazio dei nomi. Ecco come potremmo invocare la funzione build\_part:

```
part1 = build_part(528, "Disk drive", 10);
```

Sia passare una struttura a una funzione sia restituire una struttura da una funzione richiede di effettuare una copia di tutti i membri della struttura stessa. Ne risulta che queste operazioni impongono al programma una buona quantità di overhead, specialmente se la struttura è di grandi dimensioni. Per evitare questo overhead, a volte è consigliabile passare un puntatore alla struttura invece di passare la struttura stessa. Analogamente possiamo fare in modo che una funzione restituisca un puntatore a una struttura invece di restituire effettivamente la struttura. La Sezione 17.5 fornisce degli esempi di funzioni che hanno per argomenti dei puntatori a struttura e/o restituiscono dei puntatori a struttura.

Oltre all'efficienza, vi sono altre ragioni per evitare la copia delle strutture. Per esempio, l'header <stdio.h> definisce un tipo chiamato FILE [tipo FILE > 22.1], il quale, tipicamente, è una struttura. Ogni struttura FILE immagazzina delle informazioni sullo stato di un file aperto e quindi deve essere unica all'interno di un programma. Ogni funzione presente in <stdio.h> che apre un file restituisce un puntatore a una struttura FILE, e ogni funzione che esegue delle operazioni su un file aperto richiede un puntatore a FILE come argomento.

Occasionalmente potremmo voler inizializzare una variabile struttura all'interno di una funzione in modo da farla corrispondere a un'altra struttura che potrebbe essere fornita come parametro. Nell'esempio seguente l'inizializzatore per part2 è il parametro passato alla funzione f:

```
void f(struct part part1)
{
 struct part part2 = part1;
}
```

Il C permette degli inizializzatori di questo tipo, ammesso che la struttura che stiamo inizializzando (part2 in questo caso) abbia una durata di memorizzazione automatica (è locale a una funzione e non è stata dichiarata static). L'inizializzatore può essere una qualsiasi espressione del tipo appropriato, inclusa una chiamata a funzione che restituiscia una struttura.

C99

## Letterali composti

La Sezione 9.3 ha introdotto la funzionalità propria del C99 chiamata letterale composto. In quella sezione i letterali composti sono stati utilizzati per creare vettori senza nome, con lo scopo di passare un vettore a una funzione. Un letterale composto può essere usato anche per creare una struttura "al volo", senza prima memorizzarla in una variabile. La struttura risultante può essere passata come parametro, restituita da una funzione o assegnata a una variabile. Vediamo un paio di esempi.

Per prima cosa possiamo utilizzare un letterale composto per creare una struttura che verrà passata a una funzione. Per esempio, possiamo chiamare la funzione `print_part` in questo modo:

```
print_part((struct part) {528, "Disk drive", 10});
```

Il letterale composto (stampato in grassetto) crea una struttura `part` contenente nell'ordine i membri 528, "Disk drive" e 10. Questa struttura viene passata alla funzione `print_part` che si occupa di visualizzarla.

Ecco come un letterale composto potrebbe essere assegnato a una variabile:

```
part1 = (struct part) {528, "Disk drive", 10};
```

Questa istruzione somiglia a una dichiarazione contenente un inizializzatore, ma non lo è (gli inizializzatori possono comparire solamente nelle dichiarazioni e non nelle istruzioni).

In generale un letterale composto consiste di un nome di tipo racchiuso tra parentesi tonde seguito da un insieme di valori racchiusi tra parentesi graffe. Nel caso di un letterale composto che rappresenti una struttura, il nome di tipo può essere un tag di struttura preceduto dalla parola `struct` (come nei nostri esempi), oppure da un nome `typedef`. Un letterale composto può contenere dei designatori proprio come negli inizializzatori designati:

```
print_part((struct part) {.on_hand = 10,
 .name = "Disk drive",
 .number = 528});
```

Un letterale composto può non essere in grado di attuare una piena inizializzazione, in questo caso tutti i membri non inizializzati per default verranno posti a zero.

## 16.3 Annidamento di strutture e vettori

I vettori e le strutture possono essere combinati senza alcuna restrizione. I vettori possono avere delle strutture come loro elementi e le strutture possono contenere vettori e strutture come membri. Abbiamo già visto un esempio di vettori annidati all'interno

no di una struttura (il membro `name` della struttura `part`). Esploriamo le altre possibilità: strutture i cui membri sono strutture e vettori i cui elementi sono strutture.

## Strutture annidate

Spesso è utile annidare un tipo di struttura all'interno di un altro. Supponete per esempio di aver dichiarato la seguente struttura in grado di memorizzare il nome di una persona, l'iniziale del suo secondo nome e il cognome:

```
struct person_name {
 char first[FIRST_NAME_LEN+1];
 char middle_initial;
 char last[LAST_NAME_LEN+1];
};
```

Possiamo utilizzare la struttura `person_name` come parte di una struttura più grande:

```
struct student {
 struct person_name name;
 int id, age;
 char sex;
} student1, student2;
```

Accedere al nome, all'iniziale del secondo nome o al cognome di `student1` richiede un doppio utilizzo dell'operatore punto:

```
strcpy(student1.name.first, "Fred");
```

Un vantaggio di aver reso `name` una struttura (invece di avere `first`, `middle_initial` e `last` come membri della struttura `student`) è che in questo modo possiamo trattare più facilmente i nomi come unità di dato. Per esempio, se dovessimo scrivere una funzione che stampa il nome potremmo passarle solo un argomento (una struttura `person_name`) invece di tre argomenti:

```
display_name(student1.name);
```

Allo stesso modo, copiare le informazioni da una struttura `person_name` in un membro `name` di una struttura `student` richiederebbe un solo assegnamento invece di tre:

```
struct person_name new_name;
-
student1.name = new_name;
```

## Vettori di strutture

Una delle combinazioni più comuni dei vettori e delle strutture è un vettore i cui elementi sono costituiti da strutture. Un vettore di questo tipo può essere utilizzato come semplice database. Per esempio il seguente vettore di strutture `part` è in grado di memorizzare le informazioni riguardanti 100 componenti:

```
struct part inventory[100];
```

Per accedere a uno dei componenti presenti nel vettore dovremo utilizzare l'indicizzazione. Per esempio: per stampare il componente contenuto nella posizione *i* potremmo scrivere

```
print_part(inventory[i]);
```

Accedere a un membro all'interno di una struttura *part* richiede una combinazione di indicizzazione e selezione di membro. Per assegnare il valore 883 al membro *number* di *inventory[i]* dovremmo scrivere:

```
inventory[i].number = 883;
```

Accedere a un singolo carattere all'interno di un nome di un componente richiede l'indicizzazione (per selezionare il particolare componente), seguita dalla selezione (per selezionare il membro *name*), seguita dall'indicizzazione (per selezionare un carattere del nome del componente). Per modificare in una stringa vuota il nome immagazzinato in *inventory[i]*, potremo scrivere

```
inventory[i].name[0] = '\0';
```

## Inizializzare un vettore di strutture

L'inizializzazione di un vettore di strutture viene fatta praticamente allo stesso modo dell'inizializzazione di un vettore multidimensionale. Ogni struttura possiede il suo inizializzatore racchiuso tra parentesi graffe. L'inizializzatore per il vettore semplicemente racchiude tra parentesi gli inizializzatori delle strutture.

L'inizializzazione di un vettore di strutture lo rende utilizzabile come database di informazioni che non cambieranno durante l'esecuzione del programma. Per esempio, supponete di lavorare su un programma che abbia bisogno di accedere al prefisso della nazione (*country code*) quando viene effettuata una chiamata internazionale. Per prima cosa creeremo una struttura che possa contenere il nome della nazione assieme al suo prefisso:

```
struct dialing_code {
 char *country;
 int code;
};
```

Osservate che *country* è un puntatore e non un vettore di caratteri. Questo potrebbe essere un problema se stessimo pianificando di utilizzare delle strutture *dialing\_code* come variabili, tuttavia non lo stiamo facendo. Quando inizializziamo una struttura *dialing\_code*, il membro *country* finirà per puntare a una stringa letterale.

Successivamente dichiareremo un vettore di queste strutture e lo inizializzeremo per contenere i codici di alcune delle nazioni più popolose del mondo:

```
const struct dialing_code country_codes[] =
{{"Argentina", 54}, {"Bangladesh", 880},
 {"Brazil", 55}, {"Burma (Myanmar)", 95},
 {"China", 86}, {"Colombia", 57},
 {"Congo, Dem. Rep. of", 243}, {"Egypt", 20},
```

```

{"Ethiopia", 251}, {"France", 33},
 {"Germany", 49}, {"India", 91},
 {"Indonesia", 62}, {"Iran", 98},
 {"Italy", 39}, {"Japan", 81},
 {"Mexico", 52}, {"Nigeria", 234},
 {"Pakistan", 92}, {"Philippines", 63},
 {"Poland", 48}, {"Russia", 7},
 {"South Africa", 27}, {"South Korea", 82},
 {"Spain", 34}, {"Sudan", 249},
 {"Thailand", 66}, {"Turkey", 90},
 {"Ukraine", 380}, {"United Kingdom", 44},
 {"United States", 1}, {"Vietnam", 84}};

```

Le parentesi più interne attorno a ogni valore di struttura sono opzionali. Tuttavia, per questioni di stile, non le ometteremo.

A causa del fatto che i vettori di strutture (e strutture contenenti vettori) sono così comuni, gli inizializzatori designati del C99 permettono a un oggetto di avere più di un designatore. Supponete di voler inizializzare il vettore `inventory` in modo da fargli contenere un singolo componente. Il numero del componente è 528 e la quantità disponibile è 10, mentre il nome viene lasciato vuoto per ora:

```

struct part inventory[100] =
 {[0].number = 528, [0].on_hand = 10, [0].name[0] = '\0'};

```

I primi due oggetti della lista utilizzano due designatori (uno per selezionare l'elemento 0 del vettore – una struttura `part` – e uno per selezionare un membro all'interno della struttura). L'ultimo oggetto utilizza tre designatori: uno per selezionare un elemento del vettore, uno per selezionare il membro `name` di quell'elemento, e uno per selezionare l'elemento 0 di `name`.

## PROGRAMMA

## Mantenere un database di componenti

Per illustrare come i vettori e le strutture annidate vengano utilizzati nella pratica, svilupperemo un programma piuttosto lungo che mantiene un database contenente le informazioni riguardanti i componenti presenti in un magazzino. Il programma è costruito attorno a un vettore di strutture, dove ognuna di queste contiene informazioni su un componente (numero del componente, nome e quantità). Il nostro programma supporterà le seguenti operazioni.

- Aggiungere un nuovo numero di componente, nome di componente e quantità disponibile iniziale. Il programma deve stampare un messaggio di errore se il componente è già presente nel database o se il database è pieno.
- Dato un numero di componente, stampare il nome del componente e la quantità disponibile corrente. Il programma deve stampare un messaggio di errore se il numero di componente non è presente nel database.
- Dato un numero di componente, modificare la quantità disponibile. Il programma deve stampare un messaggio di errore se il numero di componente non è presente nel database.

- Stampare una tabella che mostri tutte le informazioni presenti nel database. I componenti devono essere visualizzati nell'ordine col quale sono stati inseriti.
- Terminare l'esecuzione del programma.

Per rappresentare queste operazioni utilizzeremo i codici i (insert), s (search), u (update), p (print) e q (quit). Una sessione del programma dovrebbe presentarsi in questo modo:

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10
```

```
Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10
```

```
Enter operation code: s
Enter part number: 914
Part not found.
```

```
Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5
```

```
Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2
```

```
Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8
```

| Part Number | Part Name     | Quantity on Hand |
|-------------|---------------|------------------|
| 528         | Disk drive    | 8                |
| 914         | Printer cable | 5                |

```
Enter operation code: q
```

Il programma dovrà memorizzare le informazioni relative a ogni componente in una struttura. Limiteremo le dimensioni del database a 100 componenti rendendo possibile la memorizzazione delle strutture in un vettore che chiameremo *inventory* (se questo limite dovesse rivelarsi troppo stringente, potremmo sempre cambiarlo in un secondo momento). Per tenere traccia del numero di componenti correntemente memorizzati nel vettore, utilizzeremo una variabile chiamata *num\_parts*.

Dato che il programma è controllato da un menu, è abbastanza semplice fare uno schema del ciclo principale:

```

for (;;) {
 chiede all'utente di immettere un codice operativo;
 legge il codice;
 switch (codice) {
 case 'i': esegue l'operazione di inserimento; break;
 case 's': esegue l'operazione di ricerca; break;
 case 'u': esegue l'operazione di aggiornamento; break;
 case 'p': esegue l'operazione di stampa; break;
 case 'q': termina il programma;
 default: stampa un messaggio di errore;
 }
}

```

Sarà utile creare delle funzioni separate per eseguire le operazioni di inserimento, ricerca, aggiornamento e stampa. Poiché queste funzioni dovranno accedere alla variabile `inventory` e `num_parts`, potremmo dichiararle come esterne. In alternativa possiamo dichiarare le variabili all'interno del `main` e poi passarle alle funzioni come argomenti. Dal punto di vista della progettazione di solito è meglio dichiarare le variabili come locali a una funzione piuttosto che esterne (leggete la Sezione 10.2 se vi siete dimenticati il perché). In questo programma tuttavia mettere `inventory` e `num_parts` all'interno del `main` complicherebbe ulteriormente le cose.

Per ragioni che vedremo più avanti, dividiamo il programma in tre file: `inventory.c` che conterrà la maggior parte del programma, `readline.h` che conterrà il prototipo per la funzione `read_line`, e `readline.c` che conterrà la definizione di `read_line`. Più avanti in questa sezione discuteremo gli ultimi due file, per ora ci concentreremo su `inventory.c`.

```

inventory.c /* Gestisce un database di componenti (versione vettore) */

#include <stdio.h>
#include "readline.h"

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} inventory[MAX_PARTS];

int num_parts = 0; /* il numero di componenti attualmente memorizzati */

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);

```

```

* main: chiede all'utente di immettere un codice, poi *
* chiama una funzione per eseguire l'azione *
* richiesta. Continua fino a quando l'utente non *
* immette il comando 'q'. Stampa un messaggio di *
* errore se l'utente immette un codice non ammesso. *

int main(void)
{
 char code;
 for (;;) {
 printf("Enter operation code: ");
 scanf(" %c", &code);
 while (getchar() != '\n') /* salta alla fine della riga */
 ;
 switch (code) {
 case 'i': insert();
 break;
 case 's': search();
 break;
 case 'u': update();
 break;
 case 'p': print();
 break;
 case 'q': return 0;
 default: printf("Illegal code\n");
 }
 printf("\n");
 }
}

* find_part: Cerca un componente nel vettore inventory *
* Restituisce l'indice all'interno del *
* vettore se il numero del componente viene *
* trovato, altrimenti restituisce -1 *

int find_part(int number)
{
 int i;

 for (i = 0; i < num_parts; i++)
 if (inventory[i].number == number)
 return i;
 return -1;
}

```

```

* insert: Chiede informazioni all'utente sul componente
* e poi lo inserisce nel database. Stampa un
* messaggio di errore e termina prematuramente
* nel caso in cui il componente esista già o il
* database sia pieno.

```

```
void insert(void)
{
 int part_number;

 if (num_parts == MAX_PARTS) {
 printf("Database is full; can't add more parts.\n");
 return;
 }

 printf("Enter part number: ");
 scanf("%d", &part_number);
 if (find_part(part_number) >= 0) {
 printf("Part already exists.\n");
 return;
 }

 inventory[num_parts].number = part_number;
 printf("Enter part name: ");
 read_line(inventory[num_parts].name, NAME_LEN);
 printf("Enter quantity on hand: ");
 scanf("%d", &inventory[num_parts].on_hand);
 num_parts++;
}
```

```

* search: Chiede all'utente di immettere il numero di un
* componente e poi lo cerca nel database. Se il
* componente esiste ne stampa il nome e la
* quantità disponibile, altrimenti stampa un
* messaggio di errore

```

```
void search(void)
{
 int i, number;

 printf("Enter part number: ");
 scanf("%d", &number);
 i = find_part(number);
 if (i >= 0) {
 printf("Part name: %s\n", inventory[i].name);
 printf("Quantity on hand: %d\n", inventory[i].on_hand);
 } else
 printf("Part not found.\n");
}
```

```

* update: Chiede all'utente il numero di un componente. *
* Stampa un messaggio di errore se il componente *
* non esiste, altrimenti chiede all'utente di *
* immettere la modifica alla quantità *
* disponibile e aggiorna il database. *
*****/
```

```
void update(void)
{
 int i, number, change;

 printf("Enter part number: ");
 scanf("%d", &number);
 i = find_part(number);
 if (i >= 0) {
 printf("Enter change in quantity on hand: ");
 scanf("%d", &change);
 inventory[i].on_hand += change;
 } else
 printf("Part not found.\n");
}

* print: Stampa una lista di tutti i componenti del *
* database, mostrando il numero e il nome del *
* componente e la quantità disponibile. *
* I componenti sono stampati nell'ordine in cui *
* sono stati inseriti nel database. *
*****/
```

```
void print (void)
{
 int i,
 printf("Part Number Part Name "
 "Quantity on Hand\n");
 for (i = 0; i < num_parts; i++)
 printf("%7d %-25s%11d\n",
 inventory[i].number,
 inventory[i].name, inventory[i].on_hand);
}
```

Nella funzione `main` la stringa di formato "`%c`" permette alla `scanf` di saltare gli spazi bianchi prima di leggere il codice operativo. Lo spazio nella stringa di formato è essenziale, senza di esso la `scanf` si troverebbe a volte a leggere il carattere new-line che termina la riga precedente dell'input.

Il programma contiene una funzione, `find_part`, che non viene chiamata dal `main`. Questa funzione "ausiliaria" ci aiuta a evitare del codice ridondante e a semplificare le funzioni più importanti. Chiamando `find_part`, le funzioni `insert`, `search` e `update`

possono localizzare un componente all'interno del database (o semplicemente determinare se questo componente esiste).

È rimasto solamente un ultimo dettaglio: la funzione `read_line`, che viene utilizzata dal programma per leggere il nome del componente. La Sezione 13.3 ha discusso i problemi relativi alla scrittura di una funzione di questo tipo. Sfortunatamente la versione di `read_line` di quella sezione non funzionerebbe a dovere nel nostro programma. Pensate a cosa succede quando l'utente inserisce un componente:

```
Enter part number: 528
Enter part name: Disk drive
```

L'utente preme il tasto Invio dopo aver immesso il numero del componente e lo rifa dopo averne immesso il nome. Ogni volta viene lasciato un invisibile carattere new-line che il programma deve leggere. A scopo di discussione facciamo finta che questi caratteri siano visibili:

```
Enter part number: 528¤
Enter part name: Disk drive¤
```

Quando chiamiamo la `scanf` per leggere il numero di un componente, questa "consuma" i caratteri 5, 2 e 8 mentre lascia il carattere ¤ come non letto. Se proviamo a leggere il nome del componente utilizzando la funzione `read_line` originale, questa incontrerà immediatamente il carattere ¤ e quindi fermerà la lettura. Questo problema è comune quando l'input numerico è seguito dall'input costituito da caratteri. La nostra soluzione sarà quella di scrivere una versione della `read_line` che salti lo spazio bianco prima di iniziare a salvare i caratteri. Questo non solo risolve il problema del new-line, ma ci permette anche di evitare tutti gli spazi bianchi che precedono il nome di un componente.

Dato che la `read_line` non è correlata alle altre funzioni presenti in `inventory.c` e dato che è potenzialmente riusabile in altri programmi, la scorporeremo dal file `inventory.c`. Il prototipo della `read_line` andrà nel file header `readline.h`:

```
readline.h #ifndef READLINE_H
#define READLINE_H

/*****************
 * read_line: salta i caratteri di spazio antecedenti, e
 * poi legge la parte rimanente della riga di
 * di input e la salva in str. Tronca la riga
 * se la sua lunghezza è maggiore di n.
 * Restituisce il numero di caratteri memorizzati.
*******/

int read_line(char str[], int n);
#endif
```

Metteremo la definizione di `read_line` dentro il file `readline.c`:

```
readline.c #include <ctype.h>
#include <stdio.h>
#include "readline.h"

int read_line(char str[], int n)
```

```

{
 int ch, i = 0;
 while (isspace(ch = getchar()))
 ;
 while (ch != '\n' && ch != EOF) {
 if (i < n)
 str[i++] = ch;
 ch = getchar();
 }
 str[i] = '\0';
 return i;
}

```

L'espressione

`isspace(ch = getchar())`

controlla la prima istruzione `while`. Questa espressione chiama `getchar` per leggere un carattere, salva il carattere nella variabile `ch` e poi utilizza la funzione `isspace` [funzione `isspace` > 23.5] per controllare se quest'ultimo sia o meno un carattere di spazio bianco. Se non lo è, allora il ciclo termina con `ch` contenente un carattere che non corrisponde a dello spazio bianco. La Sezione 15.3 spiega perché `ch` sia di tipo `int` invece di `char` e perché è bene fare un controllo con il valore `EOF`.

## 16.4 Unioni

Un'unione è simile a una struttura; consiste di uno o più membri che possono essere di tipo diverso. Tuttavia il compilatore alloca spazio solamente per il più grande dei membri, i quali si sovrappongono uno all'altro in questo spazio. Come risultato si ha che assegnare un nuovo valore a uno dei membri altera anche il valore degli altri.

Per illustrare le proprietà di base delle unioni, dichiariamo la variabile `unione` con due membri chiamata `u`:

```

union {
 int i;
 double d;
} u;

```

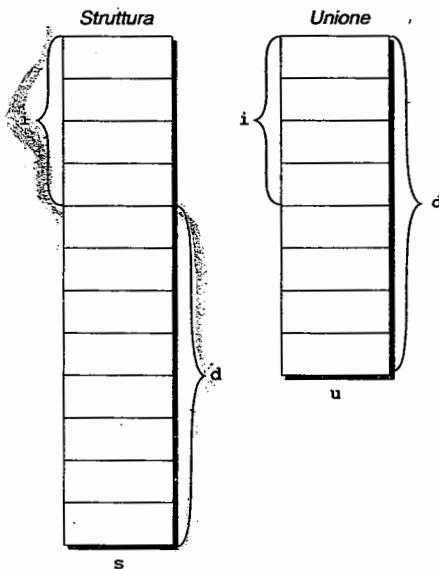
Osservate come la dichiarazione di un'unione somigli molto a quella di una struttura:

```

struct { qui di seguito
 int i;
 double d;
} s;

```

Infatti la struttura `s` e l'unione `u` differiscono solamente per il fatto che i membri di `s` sono memorizzati in indirizzi di memoria diversi, mentre i membri di `u` vengono memorizzati nello stesso indirizzo. Ecco come si presenteranno `s` e `u` nella memoria (assumendo che i valori `int` richiedano quattro byte e che i `double` ne richiedano otto):



Nella struttura s, i membri i e d occupano locazioni di memoria differenti: la dimensione totale di s è pari a 12 byte. Nell'unione u, i membri i e d si sovrappongono (i corrisponde ai primi quattro byte di d) e quindi u occupa solamente otto byte. Inoltre i e d possiedono lo stesso indirizzo.

I membri di un'unione sono accessibili allo stesso modo dei membri di una struttura. Per memorizzare il numero 82 nel membro i di u possiamo scrivere

u.i = 82;

Per salvare il valore 74.8 nel membro d, scriveremo

u.d = 74.8;

Dato che il compilatore sovrappone lo spazio di memorizzazione dei membri di un'unione, cambiare un membro altera qualsiasi valore salvato precedentemente in tutti gli altri membri. Quindi se salviamo un valore in u.d qualsiasi valore contenuto in u.i verrà perso (se esaminiamo il valore di u.i troviamo che questo è privo di significato). Analogamente, modificare il valore di u.i corrompe quello posseduto da u.d. A causa di questa proprietà possiamo pensare all'unione u come a un luogo dove memorizzare i oppure d, non entrambi (la struttura s permette di salvare i e anche d).

Le proprietà delle unioni sono praticamente identiche a quelle delle strutture. Possiamo dichiarare tag unione e tipi unione allo stesso modo nel quale dichiariamo tag e tipi struttura. Come le strutture, anche le unioni possono essere copiate tramite l'operatore =, possono essere passate alle funzioni e restituite dalle funzioni.

Le unioni possono anche essere inizializzate in modo simile a quello usato per le strutture. Tuttavia solo il primo membro di una struttura può essere impostato a un valore iniziale. Per esempio, nel modo seguente possiamo inizializzare a 0 il membro i di u:

```
union {
 int i;
 double d;
} u = {0};
```

Notate la presenza delle parentesi graffe che sono obbligatorie. L'espressione all'interno di queste parentesi deve essere costante (come vedremo nella Sezione 18.5, le regole sono leggermente diverse nel C99).

C99

Gli inizializzatori designati, una caratteristica del C99 che abbiamo discusso parlando di vettori e di strutture, possono essere utilizzati anche in abbinamento alle unioni. Un inizializzatore designato permette di specificare quale membro dell'unione debba essere inizializzato. Per esempio, possiamo inizializzare il membro *d* di *u* in questo modo:

```
union {
 int i;
 double d;
} u = {.d = 10.0};
```

Può essere inizializzato solamente un membro, ma non è necessario che sia il primo.

Ci sono diverse applicazioni delle unioni e ora ne discuteremo un paio. Altri tipi di applicazioni (come il vedere in modi diversi lo spazio di memorizzazione) sono fortemente dipendenti dalla macchina in uso e quindi li rimandiamo alla Sezione 20.3.

## Usare le unioni per risparmiare spazio

Spesso utilizzeremo delle unioni per risparmiare spazio nelle strutture. Supponete di dover progettare una struttura che andrà a contenere delle informazioni circa un articolo che viene venduto in un catalogo di regali. Il catalogo contiene solo tre tipi di merce: libri, tazze e magliette. Ogni articolo ha un numero di catalogo e un prezzo, così come altre informazioni che dipendono dal tipo di articolo:

*Libri*: Titolo, autore, numero di pagine

*Tazze*: Motivo

*Magliette*: Motivo, colori disponibili, taglie disponibili

Il nostro primo tentativo di progettazione potrebbe risultare in una struttura di questo tipo:

```
struct catalog_item {
 int stock_number;
 double price;
 int item_type;
 char title[TITLE_LEN+1];
 char author[AUTHOR_LEN+1];
 int num_pages;
 char design[DESIGN_LEN+1];
 int colors;
 int sizes;
};
```

Il membro `item_type` avrebbe uno dei seguenti valori: `BOOK`, `MUG` o `SHIRT`. I membri `colors` e `sizes` memorizzerebbero delle combinazioni codificate dei colori e delle taglie.

Sebbene questa struttura sia perfettamente utilizzabile, spreca spazio dal momento che solamente una parte delle informazioni è comune a tutti gli articoli del catalogo. Per esempio, se l'articolo è un libro non c'è bisogno di utilizzare i campi `design`, `colors` e `sizes`. Mettendo un'unione all'interno della struttura `catalog_item` possiamo ridurre lo spazio richiesto per la struttura stessa. I membri dell'unione saranno delle strutture, ognuna contenente i dati necessari per una particolare tipologia di articolo:

```
struct catalog_item {
 int stock_number;
 double price;
 int item_type;
 union {
 struct {
 char title[TITLE_LEN+1];
 char author[AUTHOR_LEN+1];
 int num_pages;
 } book;
 struct {
 char design[DESIGN_LEN+1];
 } mug;
 struct {
 char design[DESIGN_LEN+1];
 int colors;
 int sizes;
 } shirt;
 } item;
};
```

Osservate che l'unione chiamata `item` è un membro della struttura `catalog_item`, e che `book`, `mug` e `shirt` sono strutture membro di `item`. Se c'è una struttura `catalog_item` che rappresenta un libro, possiamo stampare il titolo di quest'ultimo nel modo seguente:

```
printf("%s", c.item.book.title);
```

Questo esempio dimostra che accedere a un'unione annidata dentro una struttura può essere problematico: per localizzare il titolo di un libro dobbiamo specificare il nome della struttura (`c`), il nome del membro unione della struttura (`item`), il nome di un membro struttura dell'unione (`book`) e il nome di un membro di quella struttura (`title`).

Possiamo utilizzare la struttura `catalog_item` per illustrare un aspetto interessante delle unioni. Di norma non è una buona idea memorizzare un valore all'interno di un membro di un'unione e poi accedere ai dati attraverso un membro diverso. Questo perché fare un assegnamento a un membro di un'unione fa sì che i valori degli altri membri risultino indefiniti. Tuttavia lo standard del C menziona un caso speciale, ovvero quello in cui due o più membri dell'unione sono strutture che iniziano con uno o più membri che combaciano (questi membri devono essere nello stesso ordine oltre che avere tipi compatibili, ma non devono avere necessariamente lo stesso nome). Se correntemente una delle strutture è valida allora sono validi anche i membri corrispondenti delle altre strutture.

Considerate l'unione contenuta nella struttura `catalog_item`. Questa contiene tre strutture come membro, due delle quali (`mug` e `shirt`) iniziano con un membro che combacia (`design`). Supponete ora di assegnare un valore a uno dei membri `design`:

```
strcpy(c.item.mug.design, "Cats");
```

Il membro `design` dell'altra struttura sarà definito e avrà lo stesso valore:

```
printf("%s", c.item.shirt.design); /* stampa "Cats" */
```

## Usare le unioni per creare strutture dati composite

Le unioni hanno un altro importante campo di applicazione: creare strutture dati che contengono un assortimento di dati di diverso tipo. Supponiamo di aver bisogno di un vettore i cui elementi siano un assortimento di valori `int` e `double`. Poiché gli elementi di un vettore devono essere dello stesso tipo, creare un vettore simile sembra impossibile. Tuttavia se si utilizzano le unioni è relativamente semplice. Per prima cosa definiamo un tipo unione i cui membri rappresentano i diversi tipi di dato che devono essere contenuti nel vettore:

```
typedef union {
 int i;
 double d;
} Number;
```

Successivamente creiamo un vettore i cui elementi sono valori di tipo `Number`:

```
Number number_array[1000];
```

Ogni elemento di `number_array` è un'unione `Number`. Un'unione `Number` può contenere sia un valore `int` che un valore `double` rendendo possibile il salvataggio di un assortimento di valori diversi nel vettore `number_array`. Per esempio, supponete di volere che l'elemento 0 di `number_array` contenga il valore 5, mentre l'elemento 1 contenga il valore 8.395. Gli assegnamenti seguenti produrranno l'effetto desiderato:

```
number_array[0].i = 5;
number_array[1].d = 8.395;
```

## Aggiungere un “campo etichetta” a un'unione

Le unioni presentano un problema: non c'è modo di sapere quale sia il membro che è stato modificato per ultimo e che quindi contiene un valore significativo. Considerate il problema di scrivere una funzione che visualizzi i valori correntemente memorizzati in un'unione `Number`. Questa funzione potrebbe avere questo profilo:

```
void print_number(Number n)
{
 if (n contiene un intero)
 printf("%d", n.i);
```

```

 else
 printf("%g", n.d);
}

```

Sfortunatamente la funzione `print_number` non ha modo di determinare se `n` contenga un intero o un numero a virgola mobile.

Per tenere traccia di queste informazioni possiamo includere l'unione all'interno di una struttura che possegga un altro membro: un "campo etichetta" o "discriminante", il cui scopo sia quello di ricordarci cosa è correntemente memorizzato nell'unione. Nella struttura `catalog_item` discussa precedentemente in questa sezione, il campo `item_type` serviva proprio a questo scopo.

Convertiamo il tipo `Number` in una struttura con un'unione incorporata:

```

#define INT_KIND_0
#define DOUBLE_KIND_1

typedef struct {
 int kind; /* campo etichetta */
 union {
 int i;
 double d;
 } u;
} Number;

```

`Number` possiede due membri: `kind` e `u`. Il valore di `kind` sarà uguale a `INT_KIND` o a `DOUBLE_KIND`.

Ogni volta che assegnamo un valore al membro `u` dobbiamo anche modificare `kind` per ricordarci che membro di `u` abbiamo modificato. Per esempio, se `n` è una variabile `Number`, un assegnamento al membro `i` di `u` dovrebbe presentarsi in questo modo:

```

n.kind = INT_KIND;
n.u.i = 82;

```

Osservate come l'assegnamento a `i` richieda che prima venga selezionato il membro `u` di `n` e poi il membro `i` di `u`.

Quando abbiamo bisogno di recuperare il numero memorizzato in una variabile `Number`, il membro `kind` ci dice quale membro dell'unione sia stato l'ultimo a subire un assegnamento. La funzione `print_number` può sfruttare questa possibilità:

```

void print_number(Number n)
{
 if (n.kind == INT_KIND)
 printf("%d", n.u.i);
 else
 printf("%g", n.u.d);
}

```

 È responsabilità del programma modificare il campo etichetta ogni volta che viene effettuato un assegnamento a un membro dell'unione.

## 16.5 Enumerazioni

In molti programmi avremo bisogno di variabili che possiedano solo un piccolo insieme di valori significativi. Una variabile booleana, per esempio, dovrebbe avere solo due possibili valori: "vero" e "falso". Una variabile che memorizza il seme di una carta da gioco dovrebbe possedere solo quattro possibili valori: "fiori", "quadri", "cuori" e "picche". Il modo più ovvio per gestire una variabile di questo tipo è quello di dichiararla come un intero e avere un insieme di codici rappresentanti i possibili valori che la variabile stessa può assumere:

```
int s; /* s memorizzerà un seme */
-
s = 2; /* 2 rappresenta "cuori" */
```

Sebbene questa tecnica funzioni, lascia molto a desiderare. Se qualcuno leggesse il programma non sarebbe in grado di capire che s può assumere solamente quattro possibili valori, inoltre il significato del valore 2 non sarebbe immediato.

Utilizzare delle macro per definire il "tipo" seme e i nomi dei vari semi è un passo nella direzione giusta:

```
#define SUIT int
#define CLUBS 0
#define DIAMONDS 1
#define HEARTS 2
#define SPADES 3
```

Adesso il nostro esempio precedente diventa più semplice da leggere:

```
SUIT s;
-
s = HEARTS;
```

Questa tecnica è un miglioramento, ma non è ancora la soluzione ottimale. Se qualcuno leggesse il programma non avrebbe alcuna indicazione del fatto che le macro rappresentano dei valori dello stesso "tipo". Se il numero di possibili valori non è esiguo, definire una macro diversa per ognuno di questi sarebbe tedioso. Oltre a questo, i nomi che abbiamo definito (CLUBS, DIAMONDS, HEARTS e SPADES) sarebbero rimossi dal preprocessore e quindi non sarebbero disponibili durante il debugging.

Il C fornisce uno speciale tipo adatto specificatamente alle variabili che possiedono un piccolo numero di valori ammissibili. Un tipo enumerato è un tipo i cui valori sono elencati ("enumerati") dal programmatore, il quale deve creare un nome (una costante di enumerazione) per ognuno di questi. I seguenti esempi enumerano i valori che possono essere assegnati alle variabili s1 e s2 ovvero CLUBS, DIAMONDS, HEARTS e SPADES:

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

Sebbene le enumerazioni abbiano poco in comune con le strutture e le unioni, sono dichiarate in modo simile. Tuttavia a differenza dei membri di una struttura o di una unione i nomi delle costanti di enumerazione devono essere diversi dagli altri identificatori dichiarati nello scope che li racchiude.

Le costanti di enumerazione sono simili alle costanti create con la direttiva #define, ma non sono equivalenti a queste. Il motivo è che le costanti di enumerazione sono soggette alle regole di scope del C: se un'enumerazione viene dichiarata all'interno di una funzione, le sue costanti non saranno visibili al di fuori della funzione.

## Tag e nomi di tipo di enumerazione

Spesso avremo bisogno di creare dei nomi per le enumerazioni, per la stessa ragione per la quale assegnamo un nome alle strutture e alle unioni. Come per le strutture e le unioni, ci sono due modi per dare il nome a un'enumerazione: dichiarando un tag o utilizzando typedef per creare un vero nome di tipo.

I tag di enumerazione somigliano ai tag di struttura e unione. Per definire il tag suit, per esempio, possiamo scrivere:

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

Le variabili suit verrebbero dichiarate in questo modo:

```
enum suit s1, s2;
```

In alternativa possiamo utilizzare typedef per rendere Suit un nome di tipo:

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
Suit s1, s2;
```

Nel C89 utilizzare typedef per dare il nome a un'enumerazione è un modo eccellente per creare un tipo booleano:

```
typedef enum {FALSE, TRUE} Bool;
```

Naturalmente il C99 possiede un tipo booleano nativo e quindi non vi è bisogno di definirne uno.

## Enumerazioni come gli interi

Dietro le quinte il C tratta le variabili e le costanti enumerazione come degli interi. Per default il compilatore assegna gli interi 0, 1, 2, ... alle costanti che fanno parte di una particolare enumerazione. Nella nostra enumerazione suit, per esempio, CLUBS, DIAMONDS, HEARTS e SPADES rappresentano rispettivamente i valori 0, 1, 2 e 3.

Se lo vogliamo siamo liberi di scegliere valori diversi per le costanti di enumerazione. Diciamo che CLUBS, DIAMONDS, HEARTS e SPADES corrispondono a 1, 2, 3 e 4. Possiamo specificare questi numeri quando dichiariamo l'enumerazione:

```
enum suit {CLUBS = 1, DIAMONDS = 2, HEARTS = 3, SPADES = 4};
```

I valori delle costanti di enumerazione possono essere degli interi scelti arbitrariamente e possono essere elencati senza un particolare ordine:

```
enum dept {RESEARCH = 20, PRODUCTION = 10, SALES = 25};
```

È persino ammissibile che due o più costanti di enumerazione abbiano lo stesso valore.

Quando per una costante di enumerazione non viene specificato nessun valore, questo viene posto uguale al valore della costante precedente incrementato di uno (la prima costante di enumerazione ha valore 0 per default). Nella seguente enumera-

zione BLACK possiede il valore 0, LT\_GRAY il valore 7, DK\_GRAY il valore 8 e WHITE il valore 15:

```
enum EGA_colors {BLACK, LT_GRAY = 7, DK_GRAY, WHITE = 15};
```

Dato che i valori di enumerazione non solo altro che interi leggermente camuffati, il C ci permette di mischiarli con i normali interi:

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;
i = DIAMONDS; /* i adesso vale 1 */
s = 0; /* s adesso vale 0 (CLUBS) */
s++; /* s adesso vale 1 (DIAMONDS) */
i = s + 2; /* i adesso vale 3 */
```

Il compilatore tratta s come una variabile di qualche tipo intero. CLUBS, DIAMONDS, HEARTS e SPADES sono semplicemente dei nomi per gli interi 0, 1, 2 e 3.



Sebbene la possibilità di utilizzare i valori di enumerazione come degli interi sia comoda, il contrario (utilizzare un intero come un valore di enumerazione) è pericoloso. Per esempio, potremmo salvare accidentalmente il numero 4 (che non corrisponde ad alcun simbolo) all'interno di s.

## Utilizzare le enumerazioni per dichiarare dei campi etichetta

Le enumerazioni sono perfette per risolvere il problema che abbiamo incontrato nella Sezione 16.4: determinare quale membro di un'unione sia stato l'ultimo a essere oggetto di un assegnamento. Nella struttura Number, per esempio, possiamo fare in modo che il membro kind sia un'enumerazione invece che un int:

```
typedef struct {
 enum {INT_KIND, DOUBLE_KIND} kind;
 union {
 int i;
 double d;
 } u;
} Number;
```

La nuova struttura viene utilizzata esattamente allo stesso modo di quella vecchia. I vantaggi consistono nell'esserci sbarazzati delle macro INT\_KIND e DOUBLE\_KIND (adesso sono costanti di enumerazione) e nell'aver chiarito il significato di kind (adesso è ovvio che kind può assumere solo due possibili valori: INT\_KIND e DOUBLE\_KIND).

## Domande & Risposte

**D:** Quando abbiamo provato a utilizzare l'operatore sizeof per determinare il numero di byte in una struttura, abbiamo ottenuto un numero che

**era maggiore della somma delle dimensioni dei vari membri. Come può essere?**

**R:** Guardiamo un esempio:

```
struct {
 char a;
 int b;
} s;
```

Se il valore `char` occupa un byte e i valori `int` occupano quattro byte, qual è la dimensione di `s`? La risposta ovvia (cinque byte) potrebbe non essere quella corretta. Alcuni computer richiedono che l'indirizzo di certi oggetti dato sia multiplo di un certo numero di byte (tipicamente due, quattro o otto, a seconda del tipo di oggetto). Per soddisfare questa necessità il compilatore "allinea" i membri di una struttura lasciando dei "buchi" (byte non utilizzati) tra membri adiacenti. Se assumiamo che gli oggetti dato inizino su un multiplo di quattro byte, il membro `a` della struttura `s` verrà seguito da un buco di tre byte. Come risultato si ha che `sizeof(s)` sarà uguale a 8.

Tra l'altro una struttura può avere un buco oltre che tra i membri anche alla fine di questi. Per esempio, la struttura

```
struct {
 int a;
 char b;
} s;
```

può avere un buco di tre byte dopo il membro `b`.

**D: Può esserci un "buco" all'inizio di una struttura?**

**R:** No. Lo standard C specifica che i buchi sono ammessi solo tra i membri e dopo l'ultimo di questi. Una conseguenza è che il puntatore al primo membro di una struttura è uguale al puntatore all'intera struttura (osservate però che i due puntatori non saranno dello stesso tipo).

**D: Perché non è possibile utilizzare l'operatore == per controllare se due strutture sono uguali? [p. 394]**

**R:** Questa operazione è stata lasciata al di fuori del C perché non c'è un modo di implementarla che sia coerente con la filosofia del linguaggio. Confrontare i membri di una struttura uno per uno sarebbe troppo inefficiente. Confrontare tutti i byte presenti nella struttura sarebbe una soluzione migliore (molti computer possiedono delle istruzioni speciali che possono eseguire rapidamente questo tipo di confronto). Tuttavia, se la struttura contenesse dei buchi, confrontare i byte porterebbe a un esito scorretto. Anche se membri corrispondenti avessero valori identici, i dati lasciati all'interno dei buchi potrebbero essere diversi. Il problema può essere risolto facendo in modo che il compilatore assicuri che i buchi contengano sempre lo stesso valore (diciamo lo zero). Tuttavia inizializzare i buchi imporrebbe una penalità nelle performance di tutti i programmi che utilizzano delle strutture e questo non sarebbe ammissibile.

**D: Perché il C fornisce due modi per dare un nome ai tipi struttura (tag e nomi `typedef`)? [p. 394]**

**R:** Originariamente nel C non c'era `typedef` e quindi i tag erano l'unica tecnica disponibile per dare un nome ai tipi struttura. Quando `typedef` è stato inserito era troppo tardi per rimuovere i tag. Inoltre i tag sono ancora necessari nel caso in cui un membro di una struttura punti a una struttura dello stesso tipo (guardate la struttura `node` della Sezione 17.5).

**D: Una struttura può avere sia un tag che un nome `typedef`? [p. 396]**

**R:** Sì. Infatti il tag e il nome `typedef` possono anche essere uguali, sebbene questo non sia necessario:

```
typedef struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} part;
```

**D: Come possiamo fare per condividere un tipo struttura tra i diversi file di un programma?**

**R:** Mettete una dichiarazione del tag di struttura (o un `typedef` se lo preferite) in un file header e successivamente includete quest'ultimo in tutti i file dove la struttura è necessaria. Per condividere la struttura `part`, per esempio, dovremo mettere le seguenti righe di codice in un file header:

```
struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
};
```

Osservate che stiamo dichiarando solo il *tag* di struttura e non le variabili di questo tipo.

Tra l'altro il file header che contiene la dichiarazione del tag di struttura o il tipo struttura ha bisogno di essere protetto dalle inclusioni multiple [**proteggere i file header > 15.2**]. Dichiarare due volte nello stesso file un tag o un nome `typedef` è un errore. Queste osservazioni si applicano anche alle unioni e alle enumerazioni.

**D: Se includiamo la dichiarazione della struttura `part` in due file diversi, le variabili `part` presenti in un file saranno dello stesso tipo delle variabili `part` presenti nell'altro file?**

**R:** Tecnicamente no. Tuttavia lo standard C stabilisce che le variabili `part` presenti in un file debbano essere di tipo compatibile con quelle presenti nell'altro file. Le variabili con tipo compatibile possono subire assegnamenti tra loro e quindi nella pratica c'è piccola differenza tra l'avere tipi "compatibili" e avere lo stesso tipo.

Le regole per la compatibilità delle strutture presenti nel C89 e nel C99 sono leggermente diverse. Nel C89 le strutture definite in file diversi sono compatibili se i loro membri hanno lo stesso nome e si presentano nello stesso ordine e se i membri corrispondenti sono di tipo compatibile. Il C99 va oltre: richiede che entrambe le strutture abbiano lo stesso tag o che non ce l'abbiano affatto.

Regole di compatibilità simili si applicano alle unioni e alle enumerazioni (con la stessa differenza tra il C89 e il C99).

**D: È possibile avere un puntatore a un letterale composto?**

**R:** Sì. Considerate la funzione `print_part` della Sezione 16.2. Attualmente il parametro di questa funzione è una struttura `part`. La funzione sarebbe più efficiente se venisse modificata in modo da accettare un puntatore a una struttura `part`. In tal caso per stampare un letterale composto con la funzione si dovrebbe far precedere l'argomento con l'operatore & (indirizzo):

```
print_part(&(struct part) {528, "Disk drive", 10});
```

**D: Ammettere un puntatore a un letterale composto sembra rendere possibile la modifica del letterale. È così?**

(C99)

**R:** Sì. I letterali composti sono lvalue che possono essere modificati, sebbene venga fatto raramente.

**D: Abbiamo visto un programma nel quale l'ultima costante di un'enumerazione era seguita da una virgola. Si presentava in questo modo:**

```
enum gray_values {
 BLACK = 0,
 DARK_GRAY = 64,
 GRAY = 128,
 LIGHT_GRAY = 192,
};
```

**Questa pratica è permessa?**

**R:** In effetti questa pratica è permessa nel C99 (ed è supportata anche da alcuni compilatori pre-C99). Permettere il “trascinamento” della virgola facilita la modifica delle enumerazioni perché possiamo aggiungere una costante alla fine di una enumerazione senza modificare le righe di codice esistenti. Per esempio, potremmo voler aggiungere la costante `WHITE` alla nostra enumerazione:

```
enum gray_values {
 BLACK = 0,
 DARK_GRAY = 64,
 GRAY = 128,
 LIGHT_GRAY = 192,
 WHITE = 255,
};
```

La virgola dopo la definizione di `LIGHT_GRAY` facilita l'inserimento di `WHITE` alla fine della lista.

Una ragione per questa modifica è che il C89 permette il trascinamento della virgola negli inizializzatori e quindi sembrò inconsistente non permettere la stessa flessibilità anche nelle enumerazioni. Tra l'altro il C99 permette il trascinamento della virgola anche nei letterali composti.

(C99)

**D: I valori di un tipo enumerato possono essere utilizzati anche come indici?**

R: Sì, certamente. Sono interi e hanno (per default) valori che partono dallo 0, ordinati in ordine crescente. Questo li rende degli indici perfetti. Inoltre nel C99 le costanti di enumerazione possono essere utilizzate come indici all'interno degli inizializzatori designati. Ecco un esempio:

```
enum weekdays {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
const char *daily_specials[] = {
 [MONDAY] = "Beef ravioli",
 [TUESDAY] = "BLTs",
 [WEDNESDAY] = "Pizza",
 [THURSDAY] = "Chicken fajitas",
 [FRIDAY] = "Macaroni and cheese"
};
```

## Esercizi

### Sezione 16.1

- Nelle seguenti dichiarazioni le strutture x e y possiedono dei membri chiamati x e y.

```
struct { int x, y; } x;
struct { int x, y; } y;
```

Queste dichiarazioni sono ammissibili su base individuale? Possono comparire in questo modo all'interno di un programma? Giustificate la vostra risposta.

- (a) Dichiarate delle variabili struttura chiamate c1, c2 e c3, ognuna delle quali aventi i membri real e imaginary di tipo double.  
 (b) Modificate la dichiarazione della parte (a) in modo che i membri di c1 possiedano inizialmente i valori 0.0 e 1.0, mentre i membri di c2 possiedono i valori iniziali 1.0 e 0.0 (c3 non viene inizializzata).  
 (c) Scrivete delle istruzioni che copino i membri di c2 dentro c1. Questo può essere fatto con una sola istruzione o ne richiede due?  
 (d) Scrivete delle istruzioni che sommino i membri corrispondenti di c1 e c2 salvando il risultato in c3.

### Sezione 16.2

- (a) Mostrate come dichiarare un tag chiamato complex per una struttura avente due membri, real e imaginary, di tipo double.  
 (b) Utilizzate il tag complex per dichiarare delle variabili chiamate c1, c2 e c3.  
 (c) Scrivete una funzione chiamata add\_complex che sommi i membri corrispondenti dei suoi argomenti (entrambi strutture complex) e poi restituisca il risultato della somma (un'altra struttura complex).  
 (d) Ripetete l'Esercizio 3 usando questa volta un tipo chiamato Complex.  
 (e) Scrivete le funzioni seguenti assumendo che la struttura date contenga tre membri: month, day e year (tutti di tipo int).
  - int day\_of\_year(struct date d);

Restituisce il giorno dell'anno (un intero compreso tra 1 e 366) corrispondente alla data d.

(b) int compare\_dates(struct date d1, struct date d2);

Restituisce -1 se d1 è una data precedente a d2, +1 se d1 è una data successiva a d2, 0 se d1 e d2 sono uguali.

6. Scrivete la seguente funzione assumendo che la struttura time contiene tre membri: hours, minutes e seconds (tutti di tipo int).

```
struct time split_time(long total_seconds);
```

total\_seconds è un orario rappresentato come numero di secondi a partire dalla mezzanotte. La funzione restituisce una struttura contenente l'orario equivalente in ore (0-23), minuti (0-59) e secondi (0-59).

7. Assumete che la struttura fraction contenga due numeri: numerator e denominator (entrambi di tipo int). Scrivete una funzione che esegua le seguenti operazioni sulle frazioni:

(a) Ridurre la frazione f ai minimi termini. Suggerimento: per ridurre una frazione ai minimi termini, per prima cosa calcolate il massimo comun divisore (MCD) del numeratore e del denominatore. Successivamente dividete sia il numeratore che il denominatore per il MCD.

(b) Sommare le frazioni f1 e f2.

(c) Sottrarre la frazione f2 dalla frazione f1.

(d) Moltiplicare le frazioni f1 e f2.

(e) Dividere la frazione f1 per la frazione f2.

Le frazioni f, f1 e f2 saranno degli argomenti di tipo struct fraction. Ogni funzione restituirà un valore del tipo struct fraction. Le frazioni restituite dalle funzioni presenti nei punti (b)-(e) devono essere ridotte ai minimi termini. Suggerimento: potete utilizzare la funzione del punto (a) per facilitare la scrittura delle funzioni dei punti (b)-(e).

8. Sia color la seguente struttura:

```
struct color {
 int red;
 int green;
 int blue;
};
```

(a) Scrivete la dichiarazione per una variabile const del tipo struct color chiamata MAGENTA. I membri di questa struttura dovranno avere rispettivamente i valori 255, 0, 255.

(b) (C99) Ripetete il punto (a) utilizzando un designed initializer che non specifichi il valore del membro green, facendo in modo che risulti pari a 0 per default.

9. Scrivete le funzioni seguenti (la struttura color è stata definita nell'Esercizio 8).

(a) `struct color make_color(int red, int green, int blue);`

Restituisce una struttura color contenente i valori specificati per il rosso, il verde e il blu. Se qualche argomento è minore di zero allora il membro corrispondente delle struttura viene imposto a zero. Se uno degli argomenti è maggiore di 255 allora il membro corrispondente della struttura viene imposto al valore 255.

(b) `int getRed(struct color c);`

Restituisce il valore del membro red della struttura c.

(c) `bool equal_color(struct color color1, struct color color2);`

Restituisce true se i membri corrispondenti di color1 e color2 sono uguali.

(d) `struct color brighter(struct color c);`

Restituisce una struttura color che rappresenta una versione più brillante del colore c. La struttura è identica a c ad eccezione del fatto che ogni membro è stato diviso per 0.7 (con il risultato troncato in un intero). Tuttavia ci sono tre casi speciali: (1) se tutti i membri di c sono uguali a zero, la funzione restituisce un colore i cui membri possiedono tutti il valore 3; (2) se qualche membro di c è maggiore di zero e minore di 3, allora questo viene rimpiazzato dal valore 3 prima della divisione per 0.7; (3) se dopo la divisione per 0.7 un membro diventa maggiore di 255, allora viene ridotto al valore 255.

(e) `struct color darker(struct color c);`

Restituisce una struttura color che rappresenta un versione più scura del colore c. La struttura è identica a c, ma ogni membro viene moltiplicato per 0.7 (con il risultato troncato in un intero).

**Sezione 16.3** 10. Le seguenti strutture sono state pensate per contenere delle informazioni riguardanti degli oggetti su uno schermo grafico:

`struct point { int x, y; };`

`struct rectangle { struct point upper_left, lower_right; };`

Una struttura point contiene le coordinate x e y di un punto sullo schermo. Una struttura rectangle contiene le coordinate degli angoli superiore sinistro e inferiore destro di un rettangolo. Scrivete le funzioni che eseguano le seguenti operazioni sulla struttura rectangle r la quale viene passata come argomento:

(a) Calcolare l'area di r.

(b) Calcolare il centro di r restituendo un valore point. Se la coordinata x o quella y del centro non corrisponde a un intero, allora nella struttura point deve essere memorizzata la versione troncata del valore.

(c) Spostare r di x unità nella direzione x e di y unità nella direzione y, restituendo la versione modificata di r (x e y sono degli ulteriori argomenti della funzione).

(d) Determinare se il punto p si trova all'interno di r restituendo true o false (p è un ulteriore argomento del tipo struct point).

**Sezione 16.4 11.** Supponete che s corrisponda alla seguente struttura:

```
struct {
 double a;
 union {
 char b[4];
 double c;
 int d;
 } e;
 char f[4];
} s;
```

Se i valori char occupano un byte, i valori int occupano quattro byte e i valori double occupano otto byte, quanto spazio verrà allocato per s da parte del compilatore? (Assumete che il compilatore non lasci "buchi" tra i membri).

**12.** Supponete che u corrisponda alla seguente unione:

```
union {
 double a;
 struct {
 char b[4];
 double c;
 int d;
 } e;
 char f[4];
} u;
```

Se i valori char occupano un byte, i valori int occupano quattro byte e i valori double occupano otto byte, quanto spazio verrà allocato per u da parte del compilatore? (Assumete che il compilatore non lasci "buchi" tra i membri).

**13.** Supponete che s corrisponda alla seguente struttura (point è un tag struttura dichiarato nell'Esercizio 10):

```
struct shape {
 int shape_kind; /* RECTANGLE o CIRCLE */
 struct point center; /* coordinate del centro */
 union {
 struct {
 int height, width;
 } rectangle;
 struct {
 int radius;
 } circle;
 } u;
} s;
```

Se il valore di shape\_kind è uguale a RECTANGLE, i membri height e width contengono le dimensioni di un rettangolo. Se il valore di shape\_kind è uguale a CIRCLE, il membro radius contiene il raggio di un cerchio. Indicate quali delle seguenti istruzioni sono valide e illustrate come rendere valide quelle che non lo sono:

- (a) `s.shape_kind = RECTANGLE;`
- (b) `s.center.x = 10;`
- (c) `s.height = 25;`
- (d) `s.u.rectangle.width = 8;`
- (e) `s.u.circle = 5;`
- (f) `(f) s.u.radius = 5;`

W 14. Sia `shape` il tag struttura dichiarato nell'Esercizio 13. Scrivete le funzioni, che eseguono le seguenti operazioni sulla struttura `shape s` che viene passata come argomento:

- (a) Calcolare l'area di `s`.
- (b) Spostare `s` di `x` unità nella direzione `x` e di `y` unità nella direzione `y` restituendo la versione modificata di `s` (`x` e `y` sono degli ulteriori argomenti della funzione).
- (c) Scalare `s` di un fattore `c` (un valore `double`), restituendo la versione modificata di `s` (`c` è un ulteriore argomento della funzione).

Sezione 16.5 W 15. (a) Dichiarate un tag per un'enumerazione i cui valori rappresentino i sette giorni della settimana.

(b) Utilizzate `typedef` per definire un nome per l'enumerazione del punto (a).

16. Quali delle seguenti affermazioni sulle costanti di enumerazione sono vere?

- (a) Una costante di enumerazione può rappresentare un intero specificato dal programmatore.
- (b) Le costanti di enumerazione possiedono esattamente le stesse proprietà delle costanti create usando la direttiva `#define`.
- (c) Le costanti di enumerazione per default hanno i valori `0, 1, 2, ...`
- (d) Tutte le costanti di enumerazione devono avere valori diversi.
- (e) Le costanti di enumerazione possono essere utilizzate come interi all'interno delle espressioni.

W 17. Supponete che `b` e `i` siano state dichiarate in questo modo:

```
enum {FALSE, TRUE} b;
int i;
```

Quali delle seguenti istruzioni sono ammissibili? Quali sono "sicure" (portano sempre a un risultato significativo)?

- (a) `b = FALSE;`
- (b) `b = i;`
- (c) `b++;`
- (d) `i = b;`
- (e) `i = 2 * b + 1;`

18. (a) Ogni casella di una scacchiera può contenere un pezzo (un pedone, un cavallo, un alfiere, una torre, una regina o un re) oppure può essere vuota. Ogni pezzo

può essere bianco o nero. Definite due tipi enumerati: Piece che possiede sette valori possibili (uno dei quali è "empty"), e Color che ne possiede due.

(b) Utilizzando i tipi del punto (a), definite un tipo struttura chiamato Square in grado di contenere sia il tipo di un pezzo che il suo colore.

(c) Utilizzando il tipo Square del punto (b), dichiarate un vettore  $8 \times 8$  chiamato board in grado di memorizzare l'intero contenuto di una scacchiera.

(d) Aggiungete un inizializzatore alla dichiarazione del punto (c) in modo che il valore iniziale di board corrisponda alla disposizione iniziale dei pezzi che si ha all'inizio di una partita a scacchi. Una casella non occupata da un pezzo dovrebbe possedere il valore "empty" e il colore "black".

19. Dichiarate una struttura con tag pinball\_machine che possegga i seguenti membri:

name – una stringa lunga fino a 40 caratteri.

year – un intero (rappresentante l'anno di fabbricazione).

type – un'enumerazione con i valori EM (elettromeccanico) o SS (solid state).

players – un intero (rappresentante il numero massimo di giocatori).

20. Supponete che la variabile direction venga dichiarata in questo modo:

```
enum {NORTH, SOUTH, EAST, WEST} direction;
```

Le variabili x e y sono di tipo int. Scrivete un'istruzione switch che controlli il valore di direction, incrementando x se direction è uguale a EAST, decrementando x se direction è uguale a WEST, incrementando y se direction è uguale a SOUTH e decrementando y se direction è uguale a NORTH.

21. Quali sono i valori interi delle costanti di enumerazione in ognuna delle seguenti dichiarazioni?

(a) enum {NUL, SOH, STX, ETX};

(b) enum {VT = 11, FF, CR};

(c) enum {SO = 14, SI, DLE, CAN = 24, EM};

(d) enum {ENQ = 45, ACK, BEL, LF = 37, ETB, ESC};

22. L'enumerazione chess\_piece corrisponde alle seguenti enumerazioni:

```
enum chess_pieces {KING, QUEEN, ROOK, BISHOP, KNIGHT, PAWN};
```

(a) Scrivete una dichiarazione (includendo un inizializzatore) per un vettore costante di interi chiamato piece\_value che contenga i numeri 200, 9, 5, 3, 3 e 1, rappresentanti i valori di ogni pezzo degli scacchi, dal re al pedone. In effetti il valore del re è infinito dato che la "cattura" del re (scacco matto) termina la partita, tuttavia in alcuni software del gioco degli scacchi assegnano al re un valore molto grande come 200.

(b) (C99) Ripetete il punto (a) utilizzando un inizializzatore designato per inizializzare il vettore. Utilizzate le costanti di enumerazione di chess\_pieces come indici per i designatori (Suggerimento: per un esempio guardate l'ultima domanda della Sezione D&R).

## Progetti di programmazione

- W 1. Scrivete un programma che chieda all'utente di immettere un prefisso telefonico internazionale e poi lo cerchi nel vettore `country_codes` (leggete la Sezione 16.3). Se il programma trova il prefisso, allora deve visualizzare il nome della nazione corrispondente. Se il prefisso non viene trovato, il programma deve stampare un messaggio di errore.
- W 2. Modificate il programma `inventory.c` della Sezione 16.3 in modo che l'operazione `p` (`print`) stampi i componenti ordinandoli per il numero di componente.
- W 3. Modificate il programma `inventory.c` della Sezione 16.3 facendo in modo che `inventory` e `num_part` siano locali alla funzione `main`.
- W 4. Modificate il programma `inventory.c` della Sezione 16.3 aggiungendo alla struttura `part` il membro `price`. La funzione `insert` deve chiedere all'utente il prezzo del nuovo componente. Le funzioni `search` e `print` devono visualizzare il prezzo. Aggiungete un nuovo comando che permetta all'utente di modificare il prezzo di un componente.
- W 5. Modificate il Progetto di programmazione del Capitolo 5 in modo che gli orari vengano memorizzati in un singolo vettore. Gli elementi del vettore saranno delle strutture, ognuna contenente l'orario di partenza e il corrispondente orario di arrivo (tutti gli orari saranno degli interi rappresentanti il numero di minuti dalla mezzanotte). Il programma dovrà utilizzare un ciclo per cercare nel vettore l'orario di partenza più prossimo a quello immesso dall'utente.
- W 6. Modificate il Progetto di programmazione 9 del Capitolo 5 in modo che ogni data immessa da un utente venga memorizzata in una struttura `date` (leggete l'Esercizio 5). Incorporate nel vostro programma la funzione `compare_dates` dell'Esercizio 5.

# 17 Uso avanzato dei puntatori

Nei capitoli precedenti abbiamo visto due utilizzazioni importanti dei puntatori. Il Capitolo 11 ha mostrato come l'utilizzo di un puntatore a una variabile come argomento di una funzione permette a quest'ultima di modificare la variabile stessa. Il Capitolo 12 ha mostrato come elaborare i vettori per mezzo dell'aritmetica dei puntatori.

Questo capitolo completa la trattazione dei puntatori esaminando due ulteriori campi di applicazione: l'allocazione dinamica della memoria e i puntatori a funzione.

Utilizzando l'allocazione dinamica della memoria, un programma può ottenere dei blocchi di memoria durante l'esecuzione nel preciso momento in cui ne ha bisogno. La Sezione 17.1 spiega le basi dell'allocazione dinamica della memoria. La Sezione 17.2 tratta le stringhe allocate dinamicamente; queste presentano una flessibilità maggiore rispetto ai normali vettori di caratteri. La Sezione 17.3 tratta l'allocazione dinamica della memoria in generale. La Sezione 17.4 tratta l'argomento della deallocazione della memoria (rilasciare i blocchi di memoria allocati dinamicamente quando non sono più necessari).

Le strutture allocate dinamicamente giocano un ruolo importante nella programmazione C dal momento che possono essere collegate per formare liste, alberi e altre strutture dati altamente flessibili. La Sezione 17.5 si concentra sulle liste concatenate, il tipo fondamentale di struttura dati concatenata. Una delle questioni che sorgono in questa sezione (il concetto di "puntatore a puntatore") è sufficientemente importante da richiedere una sezione a sé stante (Sezione 17.6).

La Sezione 17.7 introduce i puntatori a funzione, un concetto estremamente utile: alcune delle più potenti funzioni della libreria C richiedono dei puntatori a funzione come argomento. Esamineremo una di queste funzioni, `qsort`, che è in grado di ordinare un vettore qualsiasi.

Le ultime due sezioni discutono di funzionalità collegate ai puntatori che sono comparse per la prima volta nel C99: i puntatori restricted (Sezione 17.8) e i membri vettore flessibili (Sezione 17.9). Queste funzionalità sono principalmente di interesse dei programmati C esperti, di conseguenza entrambe queste sezioni possono essere saltate dai principianti.

## 17.1 Allocazione dinamica sulla memoria

Le strutture dati del C sono normalmente di dimensione fissa. Per esempio, il numero di elementi presente in un vettore è fissato una volta che il programma è stato compilato (nel C99 la lunghezza di un vettore a lunghezza variabile [**vettori a lunghezza variabile > 8.3**] viene determinata durante l'esecuzione del programma, ma poi rimane fissa per il resto della vita del vettore). Le strutture dati a dimensione fissa possono essere un problema: dato che siamo forzati a scegliere le loro dimensioni al momento della scrittura del programma, non possiamo modificare le dimensioni senza modificare il programma e ricompilarlo.

Considerate il programma `inventory.c` della Sezione 16.3 che permette all'utente di aggiungere degli elementi in un database di componenti. Il database viene memorizzato in un vettore di lunghezza 100. Per incrementare la capacità del database possiamo incrementare la dimensione del vettore e ricompilare il programma. Non ha importanza quanto grande facciamo il vettore, c'è sempre la possibilità di riempirlo. Fortunatamente non tutto è perduto. Il C supporta l'allocazione dinamica della memoria: la possibilità di allocare la memoria durante l'esecuzione del programma. Utilizzando l'allocazione dinamica della memoria possiamo progettare delle strutture dati che crescono (e si rimpiccoliscono) al bisogno.

Sebbene sia disponibile per tutti i tipi di dato, l'allocazione dinamica della memoria viene utilizzata più di frequente per le stringhe, i vettori e le strutture. Le strutture allocate dinamicamente sono di particolare interesse perché possiamo collegarle tra loro per creare liste, alberi e altre strutture dati.

### Funzioni di allocazione della memoria

Per allocare dinamicamente la memoria abbiamo bisogno di invocare una delle tre funzioni addette a tale scopo e che sono dichiarate nell'header `<stdlib.h>` [**header <stdlib.h> > 26.2**]:

- `malloc` – alloca un blocco di memoria ma non lo inizializza;
- `calloc` – alloca un blocco di memoria e lo azzera;
- `realloc` – ridimensiona un blocco di memoria allocato precedentemente.

Delle tre, la funzione `malloc` è la più utilizzata. Questa è più efficiente della `calloc` dato che non ha bisogno di svuotare la memoria che alloca.

Quando chiamiamo una funzione di allocazione della memoria per richiederne un blocco, questa non può conoscere il tipo di dati che stiamo pensando di inserire in tale blocco e quindi non può restituire un puntatore a un tipo ordinario come `int` o `char`. Al suo posto la funzione restituisce un valore di tipo `void *`. Un valore `void *` è un puntatore “generico” (essenzialmente solo un indirizzo di memoria).

### Puntatori nulli

Quando una funzione per l'allocazione della memoria viene invocata, c'è sempre la possibilità che questa non sia in grado di allocare un blocco di memoria sufficientemente grande da soddisfare la nostra richiesta. Se questo dovesse succedere, la funzione restituirebbe un puntatore nullo. Un puntatore nullo è un “puntatore al nulla”

(uno speciale valore che è distinguibile da tutti i puntatori validi). Dopo aver salvato il valore restituito dalla funzione in una variabile puntatore, siamo costretti a controllare se questo sia un puntatore nullo.



È responsabilità del programmatore controllare il valore restituito da una qualsiasi funzione per l'allocazione della memoria e intraprendere delle azioni adeguate se questo è un puntatore nullo. L'effetto di un tentato accesso alla memoria attraverso un puntatore nullo non è definito. Il programma può andare in crash o può comportarsi in modo inaspettato.



Il puntatore nullo è rappresentato da una macro chiamata `NULL` e quindi possiamo controllare il valore restituito dalla funzione `malloc` in questo modo:

```
p = malloc(10000);
if (p == NULL) {
 /* allocazione fallita; intraprendi azione adeguata */
}
```

Alcuni programmatori combinano assieme la chiamata alla funzione `malloc` e il test:

```
if ((p = malloc(10000)) == NULL) {
 /* allocazione fallita; intraprendi azione adeguata */
}
```



La macro `NULL` è definita in sei header: `<locale.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>` e `<time.h>` (anche l'header `<wchar.h>` del C99 definisce questa macro). Se uno di questi header è incluso nel programma allora il compilatore riconoscerà la macro `NULL`. Naturalmente un programma che utilizzi le funzioni di allocazione della memoria dovrà includere l'header `<stdlib.h>` e questo renderà disponibile la macro.

Nel C i puntatori vengono considerati true o false secondo lo stesso criterio usato per i numeri. All'interno delle condizioni, tutti i puntatori non nulli vengono considerati come veri. Solo i puntatori nulli vengono considerati come falsi. Quindi invece di scrivere

```
if (p == NULL) ...
possiamo scrivere
if (!p) ...
e invece di scrivere
if (p != NULL) ...
possiamo scrivere
if (p) ...
```

Per una questione di stile, in questo libro utilizzeremo il confronto esplicito con `NULL`.

## 17.2 Stringhe allocate dinamicamente

L'allocazione dinamica della memoria molto spesso è utile quando si lavora con le stringhe. Le stringhe vengono memorizzate in vettori di caratteri e può essere difficile pre-dire quanto questi vettori debbano essere lunghi. Allocando dinamicamente le stringhe possiamo rimandare la decisione al momento dell'esecuzione del programma.

### Utilizzare malloc per allocare memoria per una stringa

La funzione `malloc` possiede il seguente prototipo:

```
void *malloc(size_t size);
```

La funzione alloca un blocco di `size` byte e restituisce un puntatore a quest'ultimo. Osservate che `size` è di tipo `size_t` [tipo `size_t > 7.6`], ovvero un intero senza segno definito nella libreria del C. A meno di non allocare un blocco di memoria molto grande possiamo considerare `size` come un normale intero.

Utilizzare la funzione `malloc` per allocare della memoria per una stringa è facile perché il C garantisce che un valore `char` richieda esattamente un byte di memoria (in altre parole `sizeof(char)` è uguale a 1). Per allocare dello spazio per una stringa di `n` caratteri dovremo scrivere

```
p = malloc(n + 1);
```

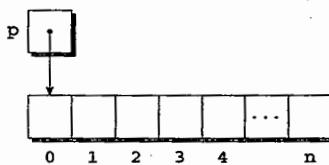
dove `p` è una variabile `char *` (l'argomento è `n+1` invece che `n` per dare spazio al carattere null). Il puntatore generico che viene restituito dalla `malloc` verrà convertito al tipo `char *` nel momento in cui viene effettuato l'assegnamento, non è necessario alcun casting (in generale possiamo assegnare un valore `*` a una variabile puntatore di qualsiasi tipo e viceversa). Nonostante ciò alcuni programmatore preferiscono effettuare il casting del valore restituito:

```
p = (char *) malloc(n + 1);
```

D&R

 Quando utilizzate la funzione `malloc` per allocare della memoria per una stringa, non dimenticatevi di includere dello spazio per il carattere null.

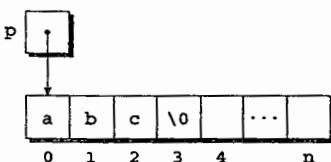
La memoria allocata usando la funzione `malloc` non è stata svuotata o inizializzata in alcun modo e quindi `p` punterà a un vettore non inizializzato di `n+1` caratteri:



Invocare la funzione `strcpy` è uno dei modi per inizializzare questo vettore:

```
strcpy(p, "abc");
```

Adesso i primi quattro caratteri del vettore sono a, b, c e \0:



## Utilizzare l'allocazione dinamica della memoria nelle funzioni per le stringhe

L'allocazione dinamica della memoria rende possibile la scrittura di funzioni che restituiscano un puntatore a una "nuova" stringa (una stringa che non esisteva prima che la funzione fosse chiamata). Considerate il problema di scrivere una funzione che concatenhi due stringhe senza modificare nessuna di queste. La libreria standard del C non include una funzione di questo tipo (strcat non corrisponde a quello che vogliamo perché modifica una delle due stringhe che le vengono passate), ma ne possiamo facilmente scrivere una nostra.

La nostra funzione misurerà la lunghezza delle due stringhe da concatenare e poi chiamerà la funzione malloc per allocare solo il giusto quantitativo di spazio necessario per contenere il risultato. La funzione successivamente copia la prima stringa nel nuovo spazio e poi chiama la funzione strcat per concatenare la seconda stringa.

```

char *concat(const char *s1, const char *s2)
{
 char *result;
 result = malloc(strlen(s1) + strlen(s2) + 1);
 if (result == NULL) {
 printf("Error: malloc failed in concat\n");
 exit(EXIT_FAILURE);
 }
 strcpy(result, s1);
 strcat(result, s2);
 return result;
}

```

Se la malloc restituisce un puntatore nullo allora la funzione concat stampa un messaggio di errore e termina il programma. Questa non è sempre l'azione giusta da intraprendere, infatti alcuni programmi hanno bisogno di riprendersi dagli insuccessi subiti nell'allocazione dinamica della memoria e continuare l'esecuzione.

Ecco un esempio di invocazione della funzione concat:

```
p = concat("abc", "def");
```

Successivamente alla chiamata, p punterà alla stringa "abcdef", la quale è contenuta in un vettore allocato dinamicamente. Il vettore è lungo sette caratteri, incluso il carattere null presente alla fine.



Le funzioni, come concat, che allocano dinamicamente la memoria, devono essere utilizzate con cautela. Quando una stringa che viene restituita dalla concat non è più necessaria, dovremo chiamare la funzione free [funzione free > 17.4] per rilasciare lo spazio occupato dalla stringa stessa. Se non lo facessimo il programma potrebbe esaurire la memoria.

## Vettori di stringhe allocate dinamicamente

Nella Sezione 13.7 abbiamo trattato il problema della memorizzazione delle stringhe all'interno di un vettore. Abbiamo visto che memorizzare le stringhe come righe di un vettore di caratteri bidimensionale può sprecare molto spazio e quindi abbiamo provato a creare un vettore di puntatori a stringhe letterali. Le tecniche della Sezione 13.7 funzionano altrettanto bene se gli elementi del vettore sono puntatori a stringhe allocate dinamicamente. Per illustrare questo punto riscriviamo il programma `remind.c` della Sezione 13.5, il quale stampa la lista di un mese di promemoria giornalieri.

### PROGRAMMA

## Stampare i promemoria di un mese (rivisitato)

Il programma `remind.c` originale salvava le stringhe dei promemoria in un vettore di caratteri bidimensionale, con ogni riga del vettore contenente una stringa. Dopo che il programma ha letto un giorno assieme al promemoria associato, effettuerà una ricerca all'interno del vettore per determinare in quale punto memorizzarlo e chiamerà la funzione `strcat` per aggiungervi il promemoria.

Nel nuovo programma (`remind2.c`), il vettore sarà unidimensionale e i suoi elementi punteranno a delle stringhe allocate dinamicamente. Convertire il programma alle stringhe allocate dinamicamente presenta principalmente due vantaggi. Il primo è il poter utilizzare lo spazio in modo più efficiente allocando l'esatto numero di caratteri necessari per contenere il promemoria, invece di salvare quest'ultimo in un numero fisso di caratteri. Secondariamente, al fine di fare spazio alle nuove stringhe, non dovremo chiamare la funzione `strcpy` per spostare quelle dei promemoria esistenti. Infatti dovremo solamente spostare dei puntatori a delle stringhe.

Ecco il nuovo programma con le modifiche in grassetto. Passare da un vettore bidimensionale a un vettore di puntatori è particolarmente semplice: dobbiamo modificare solamente otto righe di codice:

```
remind2.c /* Stampa la lista dei promemoria di un mese (versione con stringhe dinamiche)*/

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

#define MAX_REMIND 50 /* numero massimo di promemoria */
#define MSG_LEN 60 /* lunghezza massima dei messaggi */
int read_line(char str[], int n);
int main(void)
{
 char *reminders[MAX_REMIND];
 char day_str[3], msg_str[MSG_LEN+1];
 int day, i, j, num_remind = 0;
```

```
for (;;) {
 if (num_remind == MAX_REMIND) {
 printf("-- No space left --\n");
 break;
 }

 printf("Enter day and reminder: ");
 scanf("%2d", &day);
 if (day == 0)
 break;
 sprintf(day_str, "%2d", day);
 read_line(msg_str, MSG_LEN);

 for (i = 0; i < num_remind; i++)
 if (strcmp(day_str, reminders[i]) < 0)
 break;
 for (j = num_remind; j > i; j--)
 reminders[j] = reminders[j-1];

 reminders[i] = malloc(2 + strlen(msg_str) + 1);
 if (reminders[i] == NULL) {
 printf("-- No space left --\n");
 break;
 }

 strcpy(reminders[i], day_str);
 strcat(reminders[i], msg_str);

 num_remind++;
}

printf("\nDay Reminder\n");
for (i = 0; i < num_remind; i++)
 printf(" %s\n", reminders[i]);

return 0;
}
int read_line(char str[], int n)
{
 int ch, i = 0;

 while ((ch = getchar()) != '\n')
 if (i < n)
 str[i++] = ch;
 str[i] = '\0';
 return i;
}
```

## 17.3 Vettori allocati dinamicamente

I vettori allocati dinamicamente possiedono gli stessi vantaggi delle stringhe allocate dinamicamente (non è sorprendente dato che le stringhe *sono* vettori). Quando stiamo scrivendo un programma, spesso è difficile stimare la giusta dimensione per un vettore. Sarebbe molto più conveniente aspettare fino a quando il programma non viene eseguito per decidere quanto debba essere grande il vettore. Il C risolve questo problema permettendo a un programma di allocare lo spazio per un vettore durante l'esecuzione e poi accedere a quest'ultimo per mezzo di un puntatore al suo primo elemento. La stretta relazione tra i vettori e i puntatori che abbiamo esplorato nel Capitolo 12, rende i vettori allocati dinamicamente facili da usare quanto i vettori ordinari.

Sebbene la funzione `malloc` possa allocare dello spazio per un vettore, la funzione `calloc` viene utilizzata al suo posto dato che inizializza anche la memoria che alloca. La funzione `realloc` ci permette di far "crescere" o "restringere" il vettore al bisogno.

### Utilizzare `malloc` per allocare lo spazio per un vettore

Possiamo utilizzare la funzione `malloc` per allocare lo spazio per un vettore praticamente allo stesso modo utilizzato per allocare spazio per una stringa. La differenza principale risiede nel fatto che gli elementi di un qualsiasi vettore non devono essere lunghi necessariamente un byte come quelli di una stringa. Ne risulta che abbiamo bisogno di utilizzare l'operatore `sizeof` [operatore `sizeof` > 7.6] per calcolare la quantità di spazio necessaria per ogni elemento.

Supponete di scrivere un programma che necessiti di un vettore di `n` interi, dove `n` viene calcolato durante l'esecuzione del programma. Per prima cosa dichiareremo una variabile puntatore:

```
int *a;
```

Una volta conosciuto il valore `n`, il programma chiamerà la funzione `malloc` per allocare lo spazio necessario al vettore:

```
a = malloc(n * sizeof(int));
```



Utilizzate sempre l'operatore `sizeof` per calcolare quale sia lo spazio necessario per il vettore. Non allocare memoria sufficiente può portare a delle serie conseguenze. Considerate il seguente tentativo di allocare dello spazio per un vettore di `n` interi:

```
a = malloc(n * 2);
```

Se i valori `int` sono più grandi di due byte (così come succede nella maggior parte dei computer), la funzione `malloc` non allocherà un blocco di memoria sufficientemente grande. Quando in un secondo momento proviamo ad accedere agli elementi del vettore, il programma potrà andare in crash o comportarsi in modo erratico.

Una volta che punta a un blocco di memoria allocato dinamicamente, possiamo ignorare il fatto che a sia un puntatore e utilizzarlo come il nome di un vettore. Que-

sto grazie alla relazione che nel C intercorre tra i vettori e i puntatori. Per esempio, possiamo utilizzare il ciclo seguente per inizializzare il vettore puntato da a:

```
for (i = 0; i < n; i++)
 a[i] = 0;
```

Per accedere agli elementi del vettore abbiamo anche la possibilità di utilizzare l'aritmetica dei puntatori al posto dell'indicizzazione.

## La funzione calloc

Sebbene la funzione malloc possa essere utilizzata per allocare della memoria per un vettore, il C fornisce un'alternativa (la funzione calloc) che a volte risulta migliore. La funzione calloc possiede il seguente prototipo nell'header `<stdlib.h>`:

```
void *calloc(size_t nmemb, size_t size);
```

**D&R** La funzione alloca dello spazio per un vettore di `nmemb` elementi, ognuno dei quali di `size` byte. La funzione restituisce un puntatore nullo se lo spazio richiesto non è disponibile. Dopo aver allocato la memoria, la funzione calloc la inizializza impostando tutti i bit a 0. Per esempio, la chiamata seguente alla calloc alloca dello spazio per un vettore di `n` interi, i quali inizialmente sono tutti imposti a zero:

```
a = calloc(n, sizeof(int));
```

Dato che la calloc svuota la memoria che alloca, mentre la funzione malloc non lo fa, delle volte potremmo voler usare la funzione calloc per allocare dello spazio per un oggetto diverso da un vettore. Chiamando la calloc con il valore 1 come suo primo argomento, possiamo allocare dello spazio per un dato di un tipo qualsiasi:

```
struct point { int x, y; } *p;
```

```
p = calloc(1, sizeof(struct point));
```

Dopo che questa istruzione è stata eseguita, p punterà a una struttura i cui membri x e y sono imposti a zero.

## La funzione realloc

Dopo aver allocato la memoria per un vettore, potremo accorgerci che questa è troppo grande o troppo piccola. La funzione realloc può ridimensionare il vettore per adeguarsi meglio ai nostri bisogni. Il seguente prototipo per la realloc compare all'interno dell'header `<stdlib.h>`:

```
void *realloc(void *ptr, size_t size);
```

Quando la realloc viene chiamata, ptr deve puntare al blocco di memoria ottenuto dalle chiamate precedenti alle funzioni malloc, calloc o realloc. Il parametro size rappresenta la nuova dimensione del blocco, la quale può essere più grande o più piccola della dimensione originale. Sebbene la realloc non richieda che ptr punti a della memoria utilizzata come un vettore, nella pratica di solito è così.



Assicuratevi che un puntatore passato alla funzione `realloc` provenga da una chiamata precedente alle funzioni `malloc`, `calloc` o `realloc`. Se non fosse così, la chiamata alla `realloc` provocherebbe un comportamento indefinito.

Lo standard C elenca un certo numero di regole che concernono il comportamento della funzione `realloc`:

- quando espande un blocco di memoria, la `realloc` non inizializza i byte che vengono aggiunti al blocco;
- se la `realloc` non può allargare il blocco di memoria come richiesto, restituisce un puntatore nullo e i dati del vecchio blocco di memoria non vengono modificati;
- se la `realloc` viene chiamata con un puntatore nullo come primo argomento si comporta come `malloc`;
- se la `realloc` viene chiamata con 0 come suo secondo argomento, libera il blocco di memoria.

Lo standard C si sofferma brevemente a specificare il funzionamento di `realloc`. Nonostante ciò ci aspettiamo che questa sia ragionevolmente efficiente. Quando viene chiesto di ridurre la dimensione di un blocco di memoria, la funzione `realloc` deve stringerlo “nel suo posto”, ovvero senza spostare i dati contenuti al suo interno. Analogamente la funzione `realloc` deve sempre cercare di espandere il blocco di memoria senza spostarlo. Se non è in grado di allargarlo (perché i byte successivi sono già utilizzati per altri scopi), la funzione ne alloca un nuovo altrove e poi copia al suo interno i dati contenuti in quello vecchio.



Una volta che la `realloc` termina, assicuratevi di aggiornare tutti i puntatori al blocco di memoria, poiché è possibile che la funzione abbia spostato il blocco stesso.

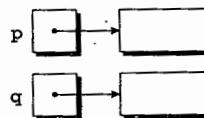
## 17.4 Deallocare la memoria

La `malloc` e le altre funzioni di allocazione della memoria ottengono dei blocchi da un’area di memoria conosciuta come heap. Chiamare queste funzioni troppo spesso (o chiedere a queste grandi blocchi di memoria) può esaurire lo heap, determinando la restituzione di un puntatore nullo da parte delle funzioni.

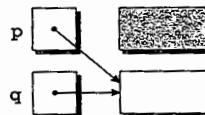
A peggiorare le cose un programma può allocare dei blocchi di memoria e poi perdere traccia di essi, sprecando così dello spazio. Considerate l’esempio seguente:

```
p = malloc(...);
q = malloc(...);
p = q;
```

Dopo l’esecuzione delle prime due istruzioni, `p` punta a un blocco di memoria mentre `q` punta a un altro:



Dopo l'assegnamento di `p` a `q`, entrambe le variabili puntano al secondo blocco di memoria:



Non ci sono puntatori al primo blocco (oscurato), di conseguenza non saremo mai più in grado di utilizzarlo.

Un blocco di memoria che non sia più accessibile da un programma viene detto garbage (spazzatura). Un programma che si lasci indietro della spazzatura si dice che è affetto da memory leak. Alcuni linguaggi forniscono un garbage collector che trova automaticamente i blocchi spazzatura e li ricicla. Il C, tuttavia, non possiede questa funzionalità. Ogni programma C è responsabile del riciclo della sua stessa spazzatura chiamando la funzione `free` per rilasciare la memoria non più necessaria.

## La funzione free

La funzione `free` ha il seguente prototipo che si trova all'interno dell'header `<stdlib.h>`:

```
void free(void *ptr);
```

Utilizzare la funzione `free` è semplice, dobbiamo semplicemente passarle un puntatore a un blocco di memoria che non è più necessario:

```
p = malloc(...);
q = malloc(...);
free(p);
p = q;
```

Chiamando la funzione `free`, questa rilascia il blocco di memoria che è puntata da `p`. Questo blocco adesso è disponibile per essere riutilizzato nelle successive chiamate alla `malloc` e alle altre funzioni di allocazione della memoria.



L'argomento alla funzione `free` deve essere un puntatore che è stato precedentemente restituito da una funzione di allocazione della memoria (l'argomento può anche essere un puntatore nullo, nel qual caso la chiamata alla `free` non ha alcun effetto). Passare alla funzione un puntatore a un qualsiasi altro oggetto (come una variabile o un elemento di un vettore) provoca un comportamento indefinito.

## Il problema del "puntatore pendente"

Sebbene la funzione free permetta di recuperare la memoria che non è più necessaria, usarla comporta un nuovo problema: il puntatore pendente. La chiamata free(p) dealloca il blocco di memoria puntato da p ma non modifica lo stesso puntatore p. Se dimentichiamo che p non punta più a un blocco di memoria valido, si può creare il caos:

```
char *p = malloc(4);
-
free(p);
-
strcpy(p, "abc"); /* SBAGLIATO */
```

Modificare la memoria puntata da p è un grave errore, poiché il nostro programma non controlla più quella porzione di memoria.



Cercare di accedere o modificare un blocco di memoria che è stato deallocated conduce a un comportamento indefinito e molto probabilmente provoca delle conseguenze disastrose tra le quali il crash del programma.

I puntatori pendenti possono essere difficili da individuare dato che diversi puntatori possono puntare allo stesso blocco di memoria. Quando un blocco viene liberato tutti i suoi puntatori vengono lasciati pendenti.

## 17.5 Liste concatenate

L'allocazione dinamica della memoria è particolarmente utile per creare liste, alberi, grafi e altre strutture dati concatenate. In questa sezione parleremo solo delle liste concatenate, una discussione delle altre strutture dati va oltre gli scopi di questo libro. Una **lista concatenata** consiste di una catena di strutture (chiamate **nodi**), ognuna delle quali contenente un puntatore al nodo successivo della catena stessa:



Nei capitoli precedenti abbiamo utilizzato un vettore ogni volta che avevamo bisogno di immagazzinare una serie di dati. Le liste concatenate ci forniscono un'alternativa. Una lista collegata è più flessibile di un vettore: possiamo facilmente inserire e cancellare nodi, permettendo alla lista di crescere o restringersi a seconda del bisogno. Per contro perdiamo la capacità di "accesso casuale" posseduta dai vettori. In un vettore si può accedere a ogni elemento nella stessa quantità di tempo. Accedere a un nodo in una lista concatenata invece è un'operazione veloce per i nodi che sono vicini all'inizio della lista, mentre è un'operazione lenta per quelli che sono prossimi alla fine.

Questa sezione descrive come creare una lista concatenata con il linguaggio C. La sezione mostra anche come eseguire le operazioni più comuni sulle liste concatenate: inserire un nodo all'inizio della lista, cercare un nodo e cancellare un nodo.

## Dichiarare un tipo nodo

Per creare una lista concatenata per prima cosa abbiamo bisogno di una struttura che rappresenti un singolo nodo della lista stessa. Assumiamo per semplicità che il nodo non contenga nulla eccetto un intero (il dato del nodo) e un puntatore al nodo successivo nella lista. Ecco come si presenterà la nostra struttura nodo:

```
struct node {
 int value; /* dato contenuto nel nodo */
 struct node *next; /* puntatore al nodo successivo */
};
```

Osservate che il membro `next` è di tipo `struct node *`, il che significa che può contenere un puntatore a una struttura `node`. Non c'è nulla di speciale nel nome `node`, è solamente un normale tag di struttura.

Un aspetto della struttura `node` merita una menzione particolare. Come è stato spiegato nella Sezione 16.2, di solito abbiamo la possibilità di scegliere se usare un tag o un nome `typedef` per definire il nome di un particolare tipo di struttura. Tuttavia, quando una struttura contiene un membro che punta a una struttura dello stesso tipo (così come fa `node`), l'utilizzo di un tag è necessario. Senza il tag della struttura `node` non avremmo modo di dichiarare il tipo del membro `next`.



Adesso che abbiamo dichiarato la struttura `node`, abbiamo bisogno di tenere traccia del punto in cui inizia la lista. In altre parole abbiamo bisogno di una variabile che punti sempre al primo nodo della lista. Chiamiamo questa variabile `first`:

```
struct node *first = NULL;
```

Imporre `first` al valore `NULL` indica che inizialmente la lista è vuota.

## Creare un nodo

Quando costruiamo una lista concatenata vogliamo creare i nodi uno alla volta e aggiungere ognuno di questi alla lista stessa. Creare un nodo richiede tre passi:

1. allocare memoria per il nodo;
2. salvare i dati nel nodo;
3. inserire il nodo nella lista.

Per ora ci concentreremo sui primi due passi.

Quando creiamo un nodo abbiamo bisogno di una variabile che punti temporaneamente a questo fino a quando non viene inserito nella lista. Chiamiamo questa variabile `new_node`:

```
struct node *new_node;
```

Utilizzeremo la funzione `malloc` per allocare la memoria per il nuovo nodo, salvando il valore restituito nella variabile `new_node`:

```
new_node = malloc(sizeof(struct node));
```

Ora `new_node` punta a un blocco di memoria grande a sufficienza per contenere una struttura `node`.



Fate attenzione a passare all'operatore `sizeof` il nome del tipo che deve essere allocato e non il nome di un puntatore a quel tipo:

```
new_node = malloc(sizeof(new_node)); /* SBAGLIATO */
```



Il programma verrà compilato comunque, tuttavia la funzione `malloc` allocherà della memoria sufficiente a contenere solo un puntatore alla struttura `node`. Il risultato più probabile è un crash del programma nel momento in cui questo cerchi di salvare dei dati all'interno del nodo al quale `new_node` avrebbe dovuto puntare.

Successivamente salveremo un dato nel membro `value` del nuovo nodo:

```
(*new_node).value = 10;
```

Ecco come si presenterà lo schema dopo l'assegnamento:



Per accedere al membro `value` del nodo abbiamo applicato l'operatore asterisco (per riferirci alla struttura puntata da `new_node`) e poi l'operatore di selezione (per selezionare uno specifico membro della struttura). Le parentesi attorno a `*new_node` sono obbligatorie a causa del fatto che l'operatore di selezione ha la precedenza rispetto all'operatore `*` [tabella degli operatori > Appendice A].

## L'operatore ->

Prima di procedere con il prossimo passo (inserire il nodo nella lista) soffermiamoci a discutere di un'utile scorciatoia. Accedere al membro di una struttura utilizzando un puntatore è una cosa così comune che il C fornisce uno speciale operatore solo per questo scopo. Questo operatore, conosciuto come freccia a destra (right arrow selection), è un segno meno seguito dal segno di maggiore. Utilizzando l'operatore `->` possiamo scrivere

```
new_node->value = 10;
```

invece di

```
(*new_node).value = 10;
```

L'operatore `->` è la combinazione dell'operatore asterisco e dell'operatore di selezione: risolve il riferimento di `new_node` per localizzare la struttura puntata dalla variabile e poi seleziona il membro `value`.

L'operatore `->` produce un lvalue [lvalue > 4.2] e quindi possiamo utilizzarlo in tutte le situazioni dove una normale variabile sarebbe ammessa. Abbiamo appena visto un esempio nel quale `new_node->value` compare nel lato sinistro di un assegnamento. L'espressione potrebbe comparire facilmente in un'invocazione alla funzione `scanf`:

```
scanf("%d", &new_node->value);
```

Osservate che l'operatore & si rivela comunque necessario anche se new\_node è un puntatore. Infatti senza di esso passeremmo alla scanf il valore di new\_node->value che è di tipo int.

## Inserire un nodo all'inizio di una lista concatenata

Uno dei vantaggi delle liste concatenate è che i nodi possono essere aggiunti in qualsiasi punto della lista: all'inizio, alla fine e in qualsiasi punto intermedio. L'inizio di una lista è il posto più semplice per inserire un nodo e quindi ci concentreremo su questo caso.

Se new\_node sta puntando al nodo che deve essere inserito e first sta puntando al primo nodo della lista concatenata, allora abbiamo bisogno di due sole istruzioni per l'inserimento. Per prima cosa modificheremo il membro next del nuovo nodo in modo che punti a quello che precedentemente era l'inizio della lista:

```
new_node->next = first;
```

Successivamente facciamo in modo che first punti al nuovo nodo:

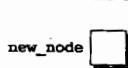
```
first = new_node;
```

Queste istruzioni funzioneranno anche nel caso in cui la lista sia vuota al momento dell'inserimento del nodo? Fortunatamente sì. Per assicurarci di questo, tracciamo il processo dell'inserimento di due nodi in una lista vuota. Inizialmente inseriremo un nodo contenente il numero 10 e poi ne inseriremo un altro contenente il valore 20. Nelle figure che seguono i puntatori nulli vengono indicati con delle linee diagonali.

```
first = NULL;
```



```
new_node = malloc(sizeof(struct node));
```



(Continua)

L'inserimento di un nodo all'interno di una lista collegata è un'operazione così comune che probabilmente merita la scrittura di una funzione dedicata a questo scopo. Chiamiamo questa funzione add\_to\_list. La funzione avrà due parametri: list (un puntatore al primo nodo della vecchia lista) e n (l'intero che deve essere salvato nel nuovo nodo).

```

new_node->value = 10;

new_node->next = first;

first = new_node;

new_node = malloc(sizeof(struct node));

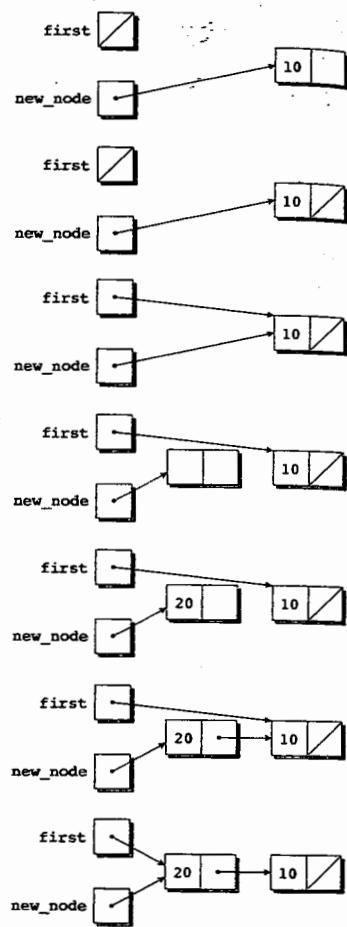
new_node->value = 20;

new_node->next = first;

first = new_node;

struct node *add_to_list(struct node *list, int n)
{
 struct node *new_node;
 new_node = malloc(sizeof(struct node));
 if (new_node == NULL) {
 printf("Error: malloc failed in add_to_list\n");
 exit(EXIT_FAILURE);
 }
 new_node->value = n;
 new_node->next = list;
 return new_node;
}

```



Notate che `add_to_list` non modifica il puntatore `list`, ma restituisce un puntatore al nuovo nodo (che adesso si trova all'inizio della lista). Quando chiamiamo `add_to_list` abbiamo bisogno di salvare il valore restituito all'interno della variabile `first`:

```
first = add_to_list(first, 10);
first = add_to_list(first, 20);
```

Queste istruzioni aggiungono dei nodi contenenti i valori 10 e 20 alla lista puntata da `first`. Fare in modo che `add_to_list` aggiorni direttamente la variabile `first` invece di restituire un nuovo valore per quest'ultima si rivelerebbe complicato. Ritorneremo su questo argomento nella Sezione 17.6.

La seguente funzione usa `add_to_list` per creare una lista concatenata contenente i numeri immessi dall'utente:

```
struct node *read_numbers(void)
{
 struct node *first = NULL;
 int n;

 printf("Enter a series of integers (0 to terminate): ");
 for (;;) {
 scanf("%d", &n);
 if (n == 0)
 return first;
 first = add_to_list(first, n);
 }
}
```

I numeri si troveranno in ordine inverso all'interno della lista dato che `first` punta sempre al nodo contenente l'ultimo valore immesso.

## Ricerca in una lista concatenata

Una volta creata una lista concatenata potremmo aver bisogno di cercare un particolare dato all'interno di essa. Sebbene per fare delle ricerche all'interno della lista si possa utilizzare il ciclo `while`, spesso l'istruzione `for` si rivela migliore. Siamo abituati a utilizzare l'istruzione `for` nella scrittura di cicli che coinvolgono un contatore, tuttavia la sua flessibilità la rende adatta anche per altri scopi, incluse le operazioni sulle liste concatenate. Ecco un modo comune per visitare i nodi di una lista concatenata utilizzando una variabile puntatore `p` per tenere traccia del nodo "corrente":

```
for (p = first; p != NULL; p = p->next)
```

L'assegnamento

```
p = p->next
```

fa avanzare il puntatore `p` da un nodo a quello successivo. Un assegnamento di questa forma viene invariabilmente usato nella scrittura di cicli che attraversano una lista concatenata.

Scriviamo una funzione chiamata `search_list` che cerca un intero `n` all'interno di una lista (puntata dal parametro `list`). Se `n` viene trovato la funzione restituisce un puntatore al nodo che lo contiene, altrimenti restituisce un puntatore nullo. La nostra prima versione di `search_list` si basa sull'idioma di "attraversamento della lista":

```
struct node *search_list(struct node *list, int n)
{
 struct node *p;
 for (p = list; p != NULL; p = p->next)
 if (p->value == n)
 return p;
 return NULL;
}
```

Naturalmente ci sono diversi modi per scrivere la funzione `search_list`. Un'alternativa sarebbe stata quella di eliminare la variabile `p` e usare al suo posto la stessa variabile `list` per tenere traccia del nodo corrente:

```
struct node *search_list(struct node *list, int n)
{
 for (; list != NULL; list = list->next)
 if (list->value == n)
 return list;
 return NULL;
}
```

Dato che l'argomento `list` è una copia del puntatore originale della lista, non si crea alcun danno a modificarlo all'interno della funzione.

Un'altra alternativa è quella di combinare il test `list->value == n` con il test `list != NULL`:

```
struct node *search_list(struct node *list, int n)
{
 for (; list != NULL && list->value != n; list = list->next)
 ;
 return list;
}
```

Dato che `list` è uguale a `NULL` quando raggiunge la fine della lista, restituire `list` è corretto anche quando il dato `n` non viene trovato. Questa versione di `search_list` può risultare un po' più chiara utilizzando l'istruzione `while`:

```
struct node *search_list(struct node *list, int n)
{
 while (list != NULL && list->value != n)
 list = list->next;
 return list;
}
```

## Eliminare un nodo da una lista concatenata

Un grande vantaggio nel memorizzare dati in una lista concatenata è dato dalla possibilità di eliminare facilmente i nodi che non sono più necessari. Eliminare un nodo come crearne uno, coinvolge tre passi:

1. localizzare il nodo che deve essere eliminato;
2. alterare il nodo precedente in modo da "bypassare" il nodo eliminato;
3. chiamare free per rilasciare lo spazio di memoria occupato dal nodo eliminato.

Il passo 1 è più complesso di quanto sembri. Se effettuiamo la ricerca all'interno della lista nel modo più ovvio, ci ritroveremo con un puntatore al nodo che deve essere cancellato. Sfortunatamente non saremo in grado di eseguire il passo 2 che richiede di modificare il nodo precedente.

Ci sono varie soluzioni al problema. Utilizzeremo la tecnica del "trascinamento del puntatore": quando effettuiamo la ricerca del passo 1 manterremo un puntatore al nodo precedente (prev) oltre che un puntatore al nodo corrente (cur). Se list punta alla lista nella quale si deve effettuare la ricerca e n è l'intero che deve essere eliminato, il ciclo seguente implementa il passo 1:

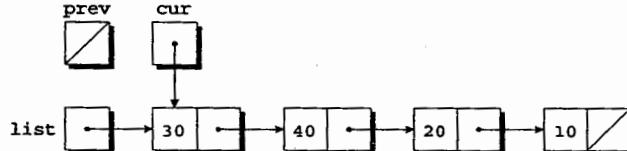
```
for (cur = list, prev = NULL;
 cur != NULL && cur->value != n;
 prev = cur, cur = cur->next)
;
```

Qui vediamo le potenzialità dell'istruzione for del C. Questo esempio piuttosto "esotico", con il corpo del ciclo vuoto e l'utilizzo dell'operatore virgola, esegue tutte le azioni necessarie per cercare n. Quando il ciclo termina, cur punta al nodo che deve essere eliminato mentre prev punta al nodo precedente (nel caso ce ne fosse uno).

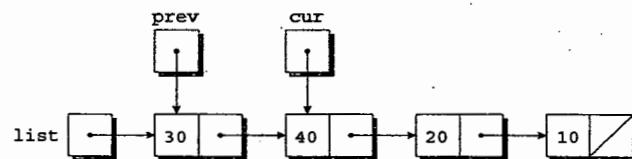
Per veder funzionare questo ciclo, assumiamo che list punti a una lista contenente 30, 40, 20 e 10 in questo ordine:



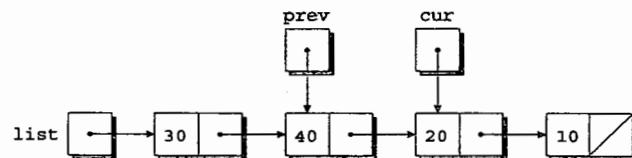
Diciamo che n è uguale a 20, quindi il nostro obiettivo è quello di eliminare il terzo nodo della lista. Dopo l'esecuzione di cur = list, prev = NULL, la variabile cur punta al primo nodo della lista:



L'espressione cur != NULL && cur->value != n è uguale a true dato che cur sta puntando a un nodo e quest'ultimo non contiene il valore 20. Dopo l'esecuzione di prev = cur, cur = cur->next, iniziamo a vedere come prev segua il percorso di cur:



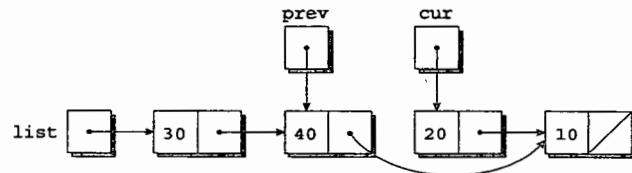
Ancora una volta l'espressione `cur != NULL && cur->value != n` è vera e quindi l'istruzione `prev = cur, cur = cur->next` viene eseguita nuovamente:



Poiché ora `cur` punta al nodo contenente 20, la condizione `cur->value != n` è falsa e quindi il ciclo termina.

Successivamente effettueremo il bypass richiesto dal passo 2. L'istruzione `prev->next = cur->next;`

fa in modo che il puntatore del nodo precedente punti al nodo successivo al nodo corrente:



Ora siamo pronti per il passo 3, ovvero rilasciare la memoria occupata dal nodo corrente:

`free(cur);`

La funzione presentata di seguito segue la strategia che abbiamo delineato. Quando vengono forniti una lista e un intero `n`, la funzione elimina il primo nodo contenente il valore `n`. Se nessun nodo contiene il valore `n`, allora la funzione non fa nulla. In entrambi i casi la funzione restituisce un puntatore alla lista.

```
struct node *delete_from_list(struct node *list, int n)
{
 struct node *cur, *prev;
 for (cur = list, prev = NULL;
 cur != NULL && cur->value != n;
 prev = cur, cur = cur->next)
 ;
```

```

 if (cur == NULL)
 return list; /* n was not found */
 if (prev == NULL)
 list = list->next; /* n is in the first node */
 else
 prev->next = cur->next; /* n is in some other node */
 free(cur);
 return list;
 }

```

Eliminare il primo nodo della lista è un caso speciale. Il test `prev == NULL` verifica se ci si trova in questa situazione, la quale richiede un'operazione di bypass diversa.

## Liste ordinate

Quando i nodi di una lista sono mantenuti in ordine (ordinati rispetto ai dati che sono contenuti all'interno dei nodi) diciamo che la lista è **ordinata**. Inserire un nodo all'interno di una lista ordinata è più difficile (il nodo non verrà inserito sempre all'inizio della lista), ma la ricerca è più veloce (possiamo fermarci dopo aver raggiunto il punto nel quale il nodo desiderato avrebbe dovuto trovarsi). Il programma seguente illustra sia l'incremento della difficoltà dovuto all'inserimento di un nodo che la maggiore velocità nella ricerca.

PROGRAMMA

## Mantenere un database di componenti (rivisitato)

Rifacciamo il programma della Sezione 16.3 relativo a un database di componenti, questa volta memorizzando il database di una lista concatenata. Utilizzare una lista concatenata al posto di un vettore presenta due vantaggi: (1) non abbiamo bisogno di stabilire un limite predefinito alla dimensione del database, questa infatti può crescere fino a quando non c'è più memoria disponibile per inserire componenti; (2) possiamo facilmente mantenere il database ordinato a seconda del numero dei componenti (quando un nuovo componente viene aggiunto a un database, inseriamo semplicemente il componente nel punto appropriato all'interno della lista). Nel programma originale il database non era ordinato.

Nel nuovo programma la struttura `part` conterrà un membro aggiuntivo (un puntatore al prossimo nodo della lista) e la variabile `inventory` diventerà un puntatore al primo nodo della lista:

```

struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
 struct part *next;
};

struct part *inventory = NULL; /* punta al primo componente */

```

La maggior parte delle funzioni del nuovo programma somiglieranno alle loro controparti del programma originale. Tuttavia le funzioni `find_part` e `insert` saranno

più complesse, dato che manteremo i nodi di inventory in una lista ordinata per numero di componente.

Nel programma originale, `find_part` restituisce un indice all'interno del vettore `inventory`. Nel nuovo programma la funzione restituirà un puntatore al nodo contenente il numero di componente desiderato. Se non trova il numero di componente, `find_part` restituirà un puntatore nullo. Dato che la lista `inventory` è ordinata per numero di componente, la nuova versione della funzione può risparmiare del tempo fermando la sua ricerca non appena trova un nodo contenente un numero di componente che è maggiore o uguale a quello desiderato. Il ciclo di ricerca di `find_part` avrà questa forma:

```
for (p = inventory;
 p != NULL && number > p->number;
 p = p->next)
;
```

Il ciclo avrà termine quando `p` diventerà uguale a `NULL` (indicando che il numero di componente non è stato trovato) o quando la condizione `number > p->number` è falsa (indicando che il numero di componente che stiamo cercando è minore o uguale al numero già contenuto in un nodo). Nell'ultimo caso ancora non sappiamo se il numero sia attualmente in una lista o meno e quindi abbiamo bisogno di un nuovo controllo:

```
if (p != NULL && number == p->number)
 return p;
```

La versione originale di `insert` salva il nuovo componente nel prossimo elemento disponibile del vettore. La nuova versione deve determinare il punto all'interno della lista nel quale inserire il nuovo componente e inserirlo. Inoltre, dobbiamo fare in modo che `insert` controlli se il numero di componente sia già presente nella lista. La funzione `insert` può occuparsi di entrambi i compiti per mezzo di un ciclo simile a quello di `find_part`:

```
for (cur = inventory, prev = NULL;
 cur != NULL && new_node->number > cur->number;
 prev = cur, cur = cur->next)
;
```

Il ciclo si basa su due puntatori: `cur`, che punta al nodo corrente, e `prev` che punta al nodo precedente. Una volta che il ciclo ha termine, la funzione `insert` controlla se il puntatore `curr` è diverso da `NULL` e se `new_node->number` è uguale a `cur->number`. Se fosse così, vorrebbe dire che il numero del componente è già presente nella lista. In caso contrario la funzione inserirà un nuovo nodo tra i nodi puntati da `prev` e `cur` utilizzando una strategia simile a quella impiegata per eliminare un nodo (questa strategia lavora anche se il nuovo componente è maggiore di tutti i componenti presenti nella lista). Ecco il nuovo programma. Come quella originale, anche questa versione utilizza la funzione `read_line` descritta nella Sezione 16.3. Assumeremo che `readline.h` contenga il prototipo di questa funzione.

```
inventory2.c /* Mantiene un database di componenti (versione con lista concatenata) */
#include <stdio.h>
```

```
#include <stdlib.h>
#include "readline.h"

#define NAME_LEN 25

struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
 struct part *next;
};

struct part *inventory = NULL; /* punta al primo componente */

struct part *find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);

/********************* main: chiede all'utente di selezionare un'operazione, *
 * poi chiama una funzione per eseguire l'azione *
 * richiesta. Continua fino a quando l'utente non *
 * immette il comando 'q'. Stampa un messaggio di *
 * errore se l'utente immette un codice non valido. *
 *****/
int main(void)
{
 char code;

 for (;;) {
 printf("Enter operation code: ");
 scanf(" %c", &code);
 while (getchar() != '\n') /* salta il fine linea */
 ;
 switch (code) {
 case 'i': insert();
 break;
 case 's': search();
 break;
 case 'u': update();
 break;
 case 'p': print();
 break;
 case 'q': return 0;
 default: printf("Illegal code\n");
 }
 printf("\n");
 }
}
```

```

* find_part:cerca nella lista un numero di componente.
* Restituisce un puntatore al nodo contenente
* il numero di componente; se il componente
* non è stato trovato, restituisce NULL.

struct part *find_part(int number)
{
 struct part *p;
 for (p = inventory;
 p != NULL && number > p->number;
 p = p->next)
 ;
 if (p != NULL && number == p->number)
 return p;
 return NULL;
}

* insert:chiede all'utente le informazioni sul nuovo
* componente e poi inserisce questo nella lista
* inventory; la lista rimane ordinata per numero
* di componente. Stampa un messaggio di errore
* e termina prematuramente se il componente
* esiste già o non è possibile allocare spazio
* per inserirlo.

void insert(void)
{
 struct part *cur, *prev, *new_node;
 new_node = malloc(sizeof(struct part));
 if (new_node == NULL) {
 printf("Database is full; can't add more parts.\n");
 return;
 }
 printf("Enter part number: ");
 scanf("%d", &new_node->number);

 for (cur = inventory, prev = NULL;
 cur != NULL && new_node->number > cur->number;
 prev = cur, cur = cur->next)
 ;
 if (cur != NULL && new_node->number == cur->number) {
 printf("Part already exists.\n");
 free(new_node);
 return;
 }
}

```

```
* * * * *
printf("Enter part name: ");
read_line(new_node->name, NAME_LEN);
printf("Enter quantity on hand: ");
scanf("%d", &new_node->on_hand);

new_node->next = cur;
if (prev == NULL)
 inventory = new_node;
else
 prev->next = new_node;
}

/*****************
 * search: chiede all'utente di inserire un numero di
 * componente e poi lo cerca nel database. Se il
 * componente esiste ne stampa il nome e la
 * quantità disponibile, altrimenti stampa un
 * messaggio di errore.
*****************/
void search(void)
{
 int number;
 struct part *p;

 printf("Enter part number: ");
 scanf("%d", &number);
 p = find_part(number);
 if (p != NULL) {
 printf("Part name: %s\n", p->name);
 printf("Quantity on hand: %d\n", p->on_hand);
 } else
 printf("Part not found.\n");
}

/*****************
 * update: chiede all'utente di immettere un numero di
 * componente. Stampa un messaggio di errore se
 * il componente non esiste, altrimenti chiede
 * all'utente di immettere la modifica della
 * quantità disponibile e aggiorna il database
*****************/
void update(void)
{
 int number, change;
 struct part *p;

 printf("Enter part number: ");
 scanf("%d", &number);
 p = find_part(number);
 if (p != NULL) {
```

```

 printf("Enter change in quantity on hand: ");
 scanf("%d", &change);
 p->on_hand += change;
 } else
 printf("Part not found.\n");
}

/********************* print *************************/
* print: stampa la lista di tutti i componenti presenti *
* nel database mostrando il numero di componente, *
* il nome e la quantità disponibile. I numeri di *
* componente compaiono in ordine crescente. *
*****void print(void)
{
 struct part *p;
 printf("Part Number Part Name
 "Quantity on Hand\n");
 for (p = inventory; p != NULL; p = p->next)
 printf("%7d %25s%11d\n", p->number, p->name,
 p->on_hand);
}

```

Osservate l'utilizzo della funzione free all'interno della funzione insert. Quest'ultima alloca della memoria per un componente prima di controllare se il componente esiste già. In quel caso la funzione rilascia lo spazio evitando la perdita di memoria.

## 17.6 Puntatori a puntatori

Nella Sezione 13.7 ci siamo imbattuti nella nozione di puntatore a puntatore e abbiamo utilizzato un vettore i cui elementi erano di tipo `char *`. Un puntatore a uno degli elementi del vettore era di tipo `char **`. Il concetto dei "puntatori a puntatori" si presenta frequentemente nel contesto delle strutture dati concatenate. In particolare quando un argomento di una funzione è una variabile puntatore, a volte desideriamo che la funzione possa modificare la variabile facendola puntare da qualche altra parte. Per fare questo utilizziamo un puntatore a un puntatore.

Considerate la funzione `add_to_list` della Sezione 17.5 che inserisce un nodo all'inizio di una lista concatenata. Quando invochiamo questa funzione, le passiamo un puntatore al primo nodo della lista originale. La funzione poi restituisce un puntatore al primo nodo della lista aggiornata:

```

struct node *add_to_list(struct node *list, int n)
{
 struct node *new_node;

 new_node = malloc(sizeof(struct node));
 if (new_node == NULL) {
 printf("Error: malloc failed in add_to_list\n");
 exit(EXIT_FAILURE);
 }

```

```

 new_node->value = n;
 new_node->next = list;
 return new_node;
}

```

Supponete di modificare la funzione in modo che, invece di restituire `new_node`, assegniate quest'ultimo a `list`. In altre parole, rimuoviamo dalla funzione l'istruzione `return` e la rimpiazziamo con

```
list = new_node;
```

Sfortunatamente questa idea non funziona. Supponete di chiamare `add_to_list` nel seguente modo:

```
add_to_list(first, 10);
```

Nel punto della chiamata, `first` viene copiato dentro `list` (i puntatori come tutti gli argomenti vengono passati per valore). L'ultima riga della funzione modifica il valore di `list` facendo in modo che punti al nuovo nodo. Questo assegnamento però non ha effetti su `first`.

Fare in modo che `add_to_list` modifichi la variabile `first` è possibile, ma richiede il passaggio alla funzione di un puntatore allo stesso `first`. Ecco la versione corretta della funzione:

```

void add_to_list(struct node **list, int n)
{
 struct node *new_node;

 new_node = malloc(sizeof(struct node));
 if (new_node == NULL) {
 printf("Error: malloc failed in add_to_list\n");
 exit(EXIT_FAILURE);
 }
 new_node->value = n;
 new_node->next = *list;
 *list = new_node;
}

```

Quando chiamiamo una nuova versione di `add_to_list`, il primo argomento sarà l'indirizzo di `first`:

```
add_to_list(&first, 10);
```

Poiché a `list` viene assegnato l'indirizzo di `first`, possiamo utilizzare `*list` come un alias per quest'ultimo. In particolare, assegnare `new_node` a `*list` andrà a modificare la variabile `first`.

## 17.7 Puntatori a funzioni

Abbiamo visto che i puntatori possono puntare a diversi tipi di dato, incluse le variabili, gli elementi dei vettori e i blocchi di memoria allocati dinamicamente. Tuttavia il C permette che i puntatori non puntino solo a *dati*, infatti c'è anche la possibilità

di avere dei puntatori alle *funzioni*. I puntatori alle funzioni non sono una cosa così bizzarra come potreste pensare. Dopo tutto le funzioni occupano delle locazioni di memoria e quindi ogni funzione possiede un suo indirizzo, proprio come ogni variabile.

## Puntatori a funzioni usati come argomenti

Possiamo utilizzare i puntatori a funzione nello stesso modo in cui utilizziamo i puntatori ai dati. In particolare, passare un puntatore a funzione come argomento è piuttosto comune in C. Supponete di dover scrivere una funzione chiamata *integrate* che integri una funzione matematica *f* tra i punti *a* e *b*. Vorremmo che la funzione *integrate* fosse la più generale possibile passandole *f* come argomento. Per ottenere questo effetto in C, dobbiamo dichiarare *f* come un puntatore a funzione. Assumendo che vogliamo integrare funzioni che abbiano un parametro *double* e restituiscano un risultato *double*, il prototipo per la funzione *integrate* si presenterà in questo modo:

```
double integrate(double (*f)(double), double a, double b);
```

Le parentesi attorno a *\*f* indicano che *f* è un puntatore a funzione e non una funzione che restituisce un puntatore. È possibile anche dichiarare *f* come se fosse una funzione:

```
double integrate(double f(double), double a, double b);
```

Dal punto di vista del compilatore questo prototipo è identico al precedente.

Quando invochiamo la *integrate*, le forniamo come primo argomento il nome di una funzione. Per esempio, la chiamata seguente integrerà la funzione *sin* (seno) [**funzione sin > 23.3**] da 0 a  $\pi/2$ :

```
result = integrate(sin, 0.0, PI / 2);
```

Osservate che dopo *sin* non ci sono parentesi. Quando il nome di una funzione non è seguito da parentesi, il compilatore C produce un puntatore alla funzione invece di generare del codice per una chiamata. Nel nostro esempio non stiamo chiamando *sin*, ma stiamo passando alla *integrate* un puntatore a *sin*. Se questo vi sembra confuso pensate a come il C tratta i vettori. Se *a* è il nome di un vettore, allora *a[1]* rappresenta un elemento del vettore, mentre lo stesso *a* costituisce un puntatore al vettore. In modo analogo, se *f* è una funzione, allora il C tratta *f(x)* come una chiamata alla funzione mentre *f* è un puntatore alla funzione stessa.

All'interno del corpo della *integrate* possiamo chiamare la funzione alla quale punta *f*:

```
y = (*f)(x);
```

*\*f* rappresenta la funzione alla quale punta *f* e *x* è l'argomento della chiamata. Quindi durante l'esecuzione di *integrate(sin, 0.0, PI/2)*, ogni chiamata a *\*f* è di fatto una chiamata alla funzione *sin*. In alternativa a *(\*f)(x)*, per chiamare la funzione puntata da *f* il C ci permette di scrivere *f(x)*. Sebbene *f(x)* appaia più naturale, noi ci attenderemo alla notazione *(\*f)(x)* per ricordarci che *f* è un puntatore a funzione e non il nome di una funzione.

## La funzione qsort

Sebbene sembri che i puntatori a funzione non siano rilevanti per il programmatore medio, ciò non può essere più lontano dalla verità. Infatti alcune delle funzioni più utili della libreria del C richiedono come argomento un puntatore a una funzione. Una di queste è la funzione `qsort` che appartiene all'header `<stdlib.h>`. La `qsort` è una funzione generica di ordinamento che è capace di ordinare qualsiasi vettore basato su qualsiasi criterio che sceglieremo.

Poiché gli elementi del vettore che ordina possono essere di qualsiasi tipo (anche strutture o unioni), la `qsort` ha bisogno che le venga detto come determinare quale tra due elementi del vettore sia il più "piccolo". Forniremo questa informazione scrivendo una **funzione di confronto**. Quando le vengono passati `p` e `q`, due puntatori a elementi del vettore, la funzione di confronto deve restituire un intero che è negativo se `*p` è "minore di" `*q`, zero se `*p` è "uguale a" `*q` e un intero positivo se `*p` è "maggiore di" `*q`. I termini "minore di", "uguale a" e "maggiore di" sono tra virgolette perché è nostra responsabilità determinare come `*p` e `*q` debbano essere confrontati.

La funzione `qsort` ha il seguente prototipo:

```
void qsort(void *base, size_t nmemb, size_t size,
 int (*compar)(const void *, const void *));
```

l'argomento `base` deve puntare al primo elemento del vettore (se deve essere ordinata solamente una porzione del vettore, faremo in modo che `base` punti al primo elemento di quella porzione). Nel caso più banale, `base` è semplicemente il nome del vettore. L'argomento `nmemb` è il numero degli elementi che devono essere ordinati (non necessariamente tutti gli elementi del vettore). L'argomento `size` è la dimensione misurata in byte di ogni elemento del vettore. L'argomento `compar` è un puntatore alla funzione di confronto. Quando la `qsort` viene chiamata, ordina il vettore in modo crescente chiamando la funzione di confronto ogni volta che ha bisogno di confrontare due elementi.

Per ordinare il vettore `inventory` della Sezione 16.3 useremo la seguente chiamata alla `qsort`:

```
qsort(inventory, num_parts, sizeof(struct part), compare_parts);
```

Osservate che il secondo argomento è `num_parts` e non `MAX_PARTS`. Infatti non vogliamo che venga ordinato l'intero vettore, ma solo la porzione nella quale sono correntemente memorizzati i componenti. L'ultimo argomento, `compare_parts` è una funzione che confronta due strutture `part`.

Scrivere la funzione `compare_parts` non è così semplice come vi potreste aspettare. La funzione `qsort` richiede che i suoi parametri siano di tipo `void *`, ma noi non possiamo accedere ai membri di una struttura `part` attraverso un puntatore `void *`, al suo posto infatti abbiamo bisogno di un puntatore del tipo `struct part *`. Per risolvere questo problema faremo in modo che `compare_parts` assegni i suoi parametri `p` e `q` a delle variabili del tipo `struct part *`, convertendoli quindi al tipo desiderato. La funzione utilizza quelle variabili per accedere ai membri delle strutture alle quali puntano `p` e `q`. Assumete di voler ordinare il vettore `inventory` secondo un ordine ascendente rispetto al numero di componente. Ecco come potrà presentarsi la funzione `compare_parts`:

```

int compare_parts(const void *p, const void *q)
{
 const struct part *p1 = p;
 const struct part *q1 = q;

 if (p1->number < q1->number)
 return -1;
 else if (p1->number == q1->number)
 return 0;
 else
 return 1;
}

```

Le dichiarazioni di p1 e q1 includono la parola const per evitare di ottenere un messaggio di warning da parte del compilatore. Poiché p e q sono puntatori const (indicando che gli oggetti ai quali puntano non devono essere modificati), devono essere assegnati solo a variabili puntatore che siano a loro volta dichiarate const.

Sebbene questa versione di compare\_parts funzionerà, la maggior parte dei programmati C la scriverebbe in modo più conciso. Per prima cosa possiamo rimpiazzare p1 e q1 con delle espressioni di casting:

```

int compare_parts(const void *p, const void *q)
{
 if (((struct part *) p)->number <
 ((struct part *) q)->number)
 return -1;
 else if (((struct part *) p)->number ==
 ((struct part *) q)->number)
 return 0;
 else
 return 1;
}

```

Le parentesi attorno a ((struct part \*) p) sono necessarie. Senza di esse il compilatore proverebbe a effettuare il cast di p->number al tipo struct part \*.

Possiamo rendere la funzione ancora più breve rimuovendo l'istruzione if:

```

int compare_parts(const void *p, const void *q)
{
 return ((struct part *) p)->number -
 ((struct part *) q)->number;
}

```

Sottraendo il numero di componente q dal numero di componente p produce un risultato negativo se p ha un numero di componente minore, un risultato pari a zero se i due numeri sono uguali e un risultato positivo se p ha un numero maggiore (osservate che sottrarre due interi è potenzialmente rischioso a causa del pericolo di overflow. Noi stiamo assumendo che il numero di componente sia un intero positivo, di conseguenza questo non dovrebbe succedere).

Per ordinare il vettore inventory per nome del componente, invece che per numero di componente, utilizzeremo la seguente versione di compare\_parts:

```
int compare_parts(const void *p, const void *q)
{
 return strcmp(((struct part *) p)->name,
 ((struct part *) q)->name);
}
```

Tutto quello che deve fare compare\_parts è chiamare la funzione strcmp, che restituisce comodamente un risultato negativo, positivo o uguale a zero.

## Altri utilizzi dei puntatori a funzione

Sebbene finora abbiamo enfatizzato l'utilità dei puntatori a funzione usati come argomenti per altre funzioni, questo non è il loro unico utilizzo. Il C tratta i puntatori a funzione esattamente come i puntatori ai dati, possiamo memorizzare i puntatori a funzione all'interno di variabili e utilizzarle come elementi di un vettore o come membri di strutture o unioni. Possiamo persino scrivere funzioni che restituiscono dei puntatori a funzione.

Ecco un esempio di una variabile che può contenere un puntatore a una funzione:

```
void (*pf)(int);
```

pf può puntare a qualsiasi funzione che abbia un parametro int e che restituisca il tipo void. Se f è una funzione di questo tipo, allora possiamo fare in modo che pf punti a f nel modo seguente:

```
pf = f;
```

Osservate come non sia stato messo nessun simbolo & davanti a f. Una volta che pf punta a f, possiamo chiamare quest'ultima sia scrivendo

```
(*pf)(i);
```

che

```
pf(i);
```

I vettori i cui elementi sono puntatori a funzione possiedono un numero sorprendente di applicazioni. Per esempio, supponete di scrivere un programma che visualizzi un menu di comandi tra i quali l'utente deve scegliere. Possiamo scrivere delle funzioni che implementano questi comandi e poi salvare nel vettore i puntatori a queste funzioni:

```
void (*file_cmd[])(void) = {new_cmd,
 open_cmd,
 close_cmd,
 close_all_cmd,
 save_cmd,
 save_as_cmd,
 save_all_cmd,
 print_cmd,
 exit_cmd
};
```

Se l'utente seleziona il comando `n`, con `n` compreso tra 0 e 8, allora possiamo inserire il vettore `file_cmd` e chiamare la funzione corrispondente:

```
(*file_cmd[n])(); /* oppure file_cmd[n](); */
```

Naturalmente avremmo ottenuto un effetto simile anche con un'istruzione `switch`. Tuttavia utilizzare un vettore di puntatori a funzione ci fornisce maggiore flessibilità che gli elementi del vettore possono essere modificati mentre il programma è in esecuzione.

#### PROGRAMMA

## Tavole delle funzioni trigonometriche

Il seguente programma mostra i valori delle funzioni `cos`, `sin` e `tan` (tutte e tre definite nell'header `<math.h>` [header `<math.h>`]). Il programma è costruito attorno a una funzione chiamata `tabulate` che, quando le viene passato un puntatore a funzione `f`, stampa una tavola contenente i valori restituiti da `f`.

```
/* Stampa una tavola dei valori delle funzioni trigonometriche */

#include <math.h>
#include <stdio.h>

void tabulate(double (*f)(double), double first,
 double last, double incr);

int main(void)
{
 double final, increment, initial;
 printf("Enter initial value: ");
 scanf("%lf", &initial);

 printf("Enter final value: ");
 scanf("%lf", &final);

 printf("Enter increment: ");
 scanf("%lf", &increment);

 printf("\n x cos(x)\n"
 "-----\n");
 tabulate(cos, initial, final, increment);

 printf("\n x sin(x)\n"
 "-----\n");
 tabulate(sin, initial, final, increment);

 printf("\n x tan(x)\n"
 "-----\n");
 tabulate(tan, initial, final, increment);

 return 0;
}
```

```
void tabulate(double (*f)(double), double first,
 double last, double incr)
{
 double x;
 int i, num_intervals;

 num_intervals = ceil((last - first) / incr);
 for (i = 0; i <= num_intervals; i++) {
 x = first + i * incr;
 printf("%10.5f %10.5f\n", x, (*f)(x));
 }
}
```

La funzione tabulate utilizza la funzione ceil, anch'essa presente in `<math.h>`. Dato un argomento `x` di tipo `double`, ceil restituisce il più piccolo intero che sia maggiore o uguale a `x`.

Ecco come potrebbe presentarsi una sessione del programma `tabulate.c`:

Enter initial value: 0

Enter final value: .5

Enter increment: .1

| x       | cos(x)  |
|---------|---------|
| 0.00000 | 1.00000 |
| 0.10000 | 0.99500 |
| 0.20000 | 0.98007 |
| 0.30000 | 0.95534 |
| 0.40000 | 0.92106 |
| 0.50000 | 0.87758 |

| x       | sin(x)  |
|---------|---------|
| 0.00000 | 0.00000 |
| 0.10000 | 0.09983 |
| 0.20000 | 0.19867 |
| 0.30000 | 0.29552 |
| 0.40000 | 0.38942 |
| 0.50000 | 0.47943 |

| x       | tan(x)  |
|---------|---------|
| 0.00000 | 0.00000 |
| 0.10000 | 0.10033 |
| 0.20000 | 0.20271 |
| 0.30000 | 0.30934 |
| 0.40000 | 0.42279 |
| 0.50000 | 0.54630 |

## 17.8 Puntatori restricted (C99)

Questa sezione e la prossima trattano due caratteristiche del C99 relative ai puntatori. Entrambe sono principalmente di interesse per i programmati C esperti, la maggior parte dei lettori vorrà saltare queste sezioni.

Nel C99 la keyword `restrict` può comparire nella dichiarazione di un puntatore:

```
int * restrict p;
```

Un puntatore che sia stato dichiarato utilizzando questa keyword viene detto **puntatore restricted**. L'intenzione è che se `p` punta a un oggetto che sia stato successivamente modificato, allora l'oggetto non sarà accessibile in altro modo che attraverso `p` (modi alternativi di accedere all'oggetto includono l'avere un altro puntatore allo stesso oggetto o l'avere `p` che punti a una variabile con nome). Avere più modi di accedere a un oggetto viene chiamato aliasing.

Guardiamo a un esempio del tipo di comportamento che i puntatori restricted dovrebbero scoraggiare. Supponete che i puntatori `p` e `q` siano stati dichiarati in questo modo:

```
int * restrict p;
int * restrict q;
```

Supponete ora che la variabile `p` venga fatta puntare a un blocco di memoria allocata dinamicamente:

```
p = malloc(sizeof(int));
```

(una situazione simile si verificherebbe se a `p` venisse assegnato l'indirizzo di una variabile o un elemento di un vettore). Normalmente sarebbe ammissibile copiare dentro `q` e poi modificare l'intero attraverso il secondo puntatore:

```
q = p;
q = 0; / provoca un comportamento indefinito */
```

A causa del fatto che `p` è un puntatore restricted, l'esecuzione dell'istruzione `*q = 0` è indefinito. Facendo sì che `p` e `q` puntino allo stesso oggetto, abbiamo fatto in modo che `*p` e `*q` siano degli alias.

Se un puntatore `p` viene dichiarato come variabile locale senza classe di memoria `extern` [classe di memorizzazione extern > 18.2], la keyword `restrict` si applica solo a `p` quando il blocco [blocchi > 10.3] nel quale viene dichiarato viene eseguito (notate che il corpo di una funzione è un blocco). La keyword `restrict` può essere utilizzata con parametri di funzione di tipo puntatore, nel qual caso si applica quando la funzione viene eseguita. Quando però `restrict` viene applicata a un puntatore con scope di file [scope di file > 10.2], la restrizione permane per tutta l'esecuzione del programma.

Le regole esatte per l'utilizzo della keyword `restrict` sono piuttosto complicate. Leggete lo standard C99 per avere maggiori dettagli. Ci sono anche delle situazioni nelle quali un alias creato a partire da un puntatore restricted è ammesso: ad esempio, è possibile copiare il puntatore restricted `p` in un'altra variabile pur

restricted q, ammesso che p sia locale alla funzione e che q sia definita all'interno di un blocco annidato dentro il corpo della funzione.

Per illustrare l'utilizzo della keyword restrict, guardiamo le funzioni `memcpy` e `memmove` che appartengono all'header `<string.h>` [header `<string.h>` > 23.6]. Nel C99 la funzione `memcpy` ha il seguente prototipo:

```
void *memcpy(void * restrict s1, const void * restrict s2,
 size_t n);
```

La funzione `memcpy` è simile alla `strcpy` a eccezione del fatto che copia i byte da un oggetto a un altro (`strcpy` copia i caratteri da una stringa a un'altra). Il parametro `s2` punta ai dati che devono essere copiati, `s1` punta alla destinazione della copia, mentre `n` è il numero di byte da copiare. L'utilizzo di `restrict` con entrambi i parametri `s1` e `s2` indica che la sorgente della copia e la destinazione non devono sovrapporsi (tuttavia non garantisce che non si sovrappongano).

Per contrasto, la keyword `restrict` non compare nel prototipo della funzione `memmove`:

```
void *memmove(void *s1, const void *s2, size_t n);
```

La funzione `memmove` effettua la stessa cosa che fa la `memcpy`: copia i byte da un posto a un altro. La differenza è che il funzionamento di `memmove` è garantito anche se la sorgente e la destinazione si sovrappongono. Per esempio, possiamo utilizzare `memmove` per far scorrere gli elementi di un vettore di una posizione:

```
int a[100];
-
memmove(&a[0], &a[1], 99 * sizeof(int));
```

Prima del C99 non c'era modo per documentare la differenza tra le funzioni `memcpy` e `memmove`. I prototipi per le due funzioni erano quasi identici:

```
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

L'uso della parola `restrict` nella versione C99 del prototipo della funzione `memcpy` fa capire al programmatore che `s1` e `s2` devono puntare degli oggetti che non si sovrappongono, altrimenti non è garantito il funzionamento della funzione.

Sebbene l'uso della keyword `restrict` nei prototipi di funzione è utile per la documentazione, questa non è la ragione principale per la sua esistenza. La keyword fornisce al compilatore delle informazioni che possono permettergli di produrre del codice più efficiente: un processo conosciuto come **ottimizzazione** (la classe di memorizzazione register serve allo stesso scopo [classe di memorizzazione register > 18.2]). Non tutti i compilatori, però, cercano di ottimizzare i programmi e quelli che normalmente lo fanno permettono al programmatore di disabilitare l'ottimizzazione. Come risultato lo standard C99 garantisce che la keyword non abbia alcun effetto sul comportamento di un programma conforme allo standard: se tutte le occorrenze della parola `restrict` venissero rimosse dal programma, questo dovrebbe comportarsi nello stesso modo.

La maggior parte dei programmatore non utilizza la keyword a meno stiano tarando finemente il programma in modo da raggiungere le migliorie possibili. In ogni caso vale la pena di conoscere `restrict` in quanto nei prototipi C99 di molte funzioni della libreria standard.

## 17.9 Membri vettore flessibili (C99)

A volte avremo bisogno di definire una struttura contenente un vettore di stringhe sconosciuta. Per esempio, potremmo voler salvare delle stringhe in un'area che è diversa da quella usuale. Normalmente una stringa è costituita da un array di caratteri, con un carattere null che ne segnala la fine. Tuttavia ci sono dei vari modi per memorizzare le stringhe in modi diversi da questo. Un'alternativa è quella di memorizzare la lunghezza della stringa assieme ai caratteri della stringa stessa (ma senza il carattere null). La lunghezza e i caratteri possono essere memorizzati in una struttura come questa:

```
struct vstring {
 int len;
 char chars[N];
};
```

Nel codice `N` è una macro che rappresenta la lunghezza massima della stringa. Il problema è che per memorizzare un vettore di lunghezza prefissata come questo è poco raccomandabile, poiché costringe a limitare la lunghezza della stringa e inoltre spreca della memoria, dato che la maggior parte delle stringhe non avranno bisogno di tutti gli `N` caratteri del vettore.

Tradizionalmente i programmatore C hanno risolto il problema dichiarando la lunghezza dei caratteri pari a 1 (un valore fittizio) e poi allocando dinamicamente uno spazio per ogni stringa:

```
struct vstring {
 int len;
 char chars[1];
};

struct vstring *str = malloc(sizeof(struct vstring) + n - 1);
str->len = n;
```

Stiamo "barando" in quanto allochiamo più memoria di quanta la struttura richiede, chiari di avere (in questo caso  $n - 1$  caratteri in più) e utilizziamo la memoria per contenere gli elementi aggiuntivi del vettore `chars`. Questa tecnica è diventata comune negli anni che le è stato dato un nome: "struct hack".

Lo struct hack non si limita ai vettori di caratteri, ma ha un gran numero di applicazioni. Nel tempo è divenuta una tecnica così popolare da essere supportata da molti compilatori. Alcuni (tra cui GCC) ammettono persino che il vettore `chars` abbia una lunghezza zero, il che rende questo trucco più esplicito. Sfortunatamente lo standard C89 non garantisce il funzionamento di questa tecnica e nemmeno permette di utilizzarla.

Come riconoscimento dell'utilità della tecnica dello struct hack, il C99 possiede una caratteristica conosciuta come **membro vettore flessibile** (*flexible array member*) che serve a questo scopo. Quando l'ultimo membro di una struttura è un vettore, la sua lunghezza può essere omessa:

```
struct vstring {
 int len;
 char chars[]; /* membro vettore flessibile - solo C99 */
};
```

La lunghezza del vettore chars non è determinata fino a che la memoria non viene allocata per una struttura vstring. Normalmente questo avviene invocando la funzione malloc:

```
struct vstring *str = malloc(sizeof(struct vstring) + n);
str->len = n;
```

In questo esempio, str punta a una struttura vstring nella quale il vettore chars occupa n caratteri. L'operatore sizeof ignora il membro chars quando calcola la dimensione della struttura (un membro vettore flessibile è inusuale nel fatto che non occupa spazio all'interno della struttura).

A una struttura contenente un membro vettore flessibile si applicano alcune regole speciali. I membri vettore flessibili devono comparire per ultimi nelle strutture e queste devono avere almeno un altro membro. Copiare una struttura contenente un membro vettore flessibile copierà gli altri membri ma non il vettore flessibile.

Una struttura che contenesse un membro vettore flessibile è un **tipo incompleto**. A un tipo incompleto manca quella parte di informazione necessaria per determinare quanta memoria richieda. I tipi incompleti, che sono trattati in una delle domande della Sezione Domande & Risposte alla fine del capitolo e nella Sezione 19.3, sono soggetti a varie restrizioni. In particolare un tipo incompleto (e quindi una struttura contenente un membro vettore flessibile) non può essere il membro di un'altra struttura o l'elemento di un vettore. Tuttavia un vettore può contenere dei puntatori a delle strutture che possiedono un membro vettore flessibile. Il Progetto di programmazione 7 alla fine di questo capitolo è costruito attorno a questo tipo di vettori.

## Domande & Risposte

**D: Cosa rappresenta la macro NULL? [p. 429]**

**R:** Agli effetti pratici NULL corrisponde al valore 0. Quando utilizziamo lo 0 in un contesto dove sarebbe richiesto un puntatore, i compilatori C tratteranno questo come un puntatore nullo invece che come l'intero 0. La macro NULL è prevista solo per evitare confusione. L'assegnamento

```
p = 0;
```

può essere l'assegnamento del valore 0 a una variabile numerica o l'assegnamento di un puntatore nullo a una variabile puntatore: non possiamo dire facilmente a quale delle due situazioni si riferisca l'assegnamento. Al contrario nell'assegnamento

`p = NULL;`

è chiaro che la variabile `p` è un puntatore.

**\*D:** Nei file di header associati al mio compilatore la macro `NULL` viene definita in questo modo:

```
#define NULL (void *) 0
```

**Qual è il vantaggio di effettuare il cast di 0 al tipo void \*?**

**D:** Questo è un trucco permesso dallo standard C che dà la possibilità ai compilatori di individuare usi errati dei puntatori nulli. Per esempio, supponete di pro-

assegnare `NULL` a una variabile intera:

`i = NULL;`

Se la macro `NULL` fosse definita uguale a 0, questo assegnamento sarebbe perfettamente ammissibile. Se invece `NULL` viene definita come `(void *) 0`, il compilatore può notarci del fatto che stiamo assegnando un puntatore a una variabile intera.

Definire `NULL` come `(void *) 0` possiede un secondo importante vantaggio: ponete di chiamare una funzione con un elenco di argomenti di lunghezza variabile [elenchi di argomenti a lunghezza variabile > 26.1] e di passarle la macro `NULL` come argomento. Se `NULL` è definita come uguale a 0, allora il compilatore passerà automaticamente un valore intero pari a zero (in una normale chiamata di funzione, la macro `NULL` funziona correttamente perché il compilatore sa che il prototipo di funzione aspetta un puntatore. Quando però una funzione ha un elenco di argomenti a lunghezza variabile, il compilatore non ha queste informazioni). Se la macro `NULL` è definita come `(void*) 0`, il compilatore passerà un puntatore nullo.

A rendere le cose ancora più confuse c'è il fatto che alcuni file header definiscono la macro `NULL` uguale a `0L` (la versione long di 0). Questa definizione, come la definizione di `NULL` uguale a 0, è un residuo dei primi anni del C, quando i puntatori interi erano compatibili. Tuttavia per la maggior parte degli scopi comuni, nessuna importanza come sia stata definita la macro `NULL`: pensate a essa semplicemente come il nome del puntatore nullo.

**D: Dato che lo 0 viene utilizzato per rappresentare il puntatore nullo, immagino che quest'ultimo sia semplicemente un indirizzo con tutti gli uguali a zero, giusto?**

**R:** Non necessariamente. Ai compilatori C è permesso di rappresentare i puntatori nulli in modo diverso e non tutti i compilatori utilizzano un indirizzo uguale a 0. Per esempio alcuni compilatori utilizzano un indirizzo di memoria inesistente per rappresentare un puntatore nullo. In questo modo ogni tentativo di accedere alla memoria attraverso un puntatore nullo può essere rilevato dall'hardware.

Come il puntatore nullo venga memorizzato all'interno del computer non riguarda nulla; questo è un dettaglio del quale devono preoccuparsi solo gli esperti di compilatori. La cosa importante è che, quando viene utilizzato in un contesto che riguarda ai puntatori, lo 0 viene convertito dal compilatore nel formato interno appropriato.

**D: È possibile utilizzare la macro `NULL` come carattere null?**

**R:** Assolutamente no. `NULL` è una macro che rappresenta un puntatore nullo come carattere null. Utilizzare `NULL` come carattere null funzionerà con alcuni compilatori,

ma non con tutti (dato che alcuni definiscono NULL come `(void*) 0`). In ogni caso utilizzare NULL in modo diverso che come puntatore può provocare un sacco di confusione. Se volete dare un nome al carattere null, definite questa macro:

```
#define NUL '\0'
```

**\*D: Quando il nostro programma termina otteniamo il messaggio "Null pointer assignment". Cosa significa?**

**R:** Questo messaggio viene prodotto dai programmi compilati con qualche vecchio compilatore basato sul DOS e indica che il programma ha salvato dei dati in memoria utilizzando un puntatore non corretto (non necessariamente un puntatore nullo). Sfortunatamente il messaggio non viene visualizzato fino al termine del programma e quindi non c'è alcun indizio di quale istruzione lo abbia causato. Il messaggio "Null pointer assignment" può essere causato da un `&` mancante in una chiamata alla `scanf`:

```
scanf("%d", i); /* avrebbe dovuto essere scanf("%d", &i); */
```

Un'altra possibilità è un assegnamento che coinvolge un puntatore non inizializzato o nullo:

```
p = i; / p è nullo o non inizializzato */
```

**\*D: Come fa un programma a sapere che si è verificato un "Null pointer assignment"?**

**R:** Il messaggio dipende dal fatto che nei modelli di memoria piccoli e medi, i dati vengono memorizzati in un singolo segmento con un indirizzo che inizia a 0. Il compilatore lascia uno spazio all'inizio del segmento dati (un piccolo blocco di memoria che è inizializzato a 0 ma che non viene utilizzato altrimenti dal programma). Quando il programma termina, controlla se nell'area corrispondente a tale spazio ci sono dei dati diversi da zero. In tal caso l'area deve essere stata alterata attraverso un puntatore sbagliato.

**D: C'è qualche vantaggio nel casting del valore restituito dalla funzione malloc o dalle altre funzioni di allocazione della memoria? [p. 430]**

**R:** Di solito no. Il casting del puntatore che viene restituito da queste funzioni non è necessario dato che il tipo `void *` viene convertito automaticamente in qualsiasi altro tipo di puntatore durante un assegnamento. L'abitudine di effettuare il casting del valore restituito è un residuo delle vecchie versioni del C, nelle quali le funzioni per l'allocazione della memoria restituivano un valore `char *`, il che rendeva il casting necessario. I programmi che sono pensati per essere compilati come codice C++ possono beneficiare del casting, ma questa è l'unica ragione per farlo.

Effettivamente nel C89 c'è un piccolo vantaggio nel non eseguire il casting. Supponete di aver dimenticato di includere nel programma l'header `<stdlib.h>`. Quando invochiamo la funzione `malloc`, il compilatore assumerà che il suo valore restituito sia di tipo `int` (il valore restituito di default da ogni funzione C). Se non effettuiamo il cast del valore restituito dalla `malloc`, un compilatore C89 produrrà un messaggio di errore (o un warning) dato che stiamo cercando di assegnare un valore intero a una variabile puntatore. D'altra parte se effettuiamo il casting, il programma potrà compilare anche se molto probabilmente non funzionerà a dovere. Con il C99 questo vantaggio scompare. Dimenticare di includere l'header `<stdlib.h>` provocherà un

errore quando malloc viene chiamata perché il C99 richiede che una funzione dichiarata prima di essere chiamata.

**D: La funzione calloc inizializza un blocco di memoria impostando tutti i bit a zero. Questo significa che tutti i dati nel blocco diventeranno a zero? [p. 435]**

R: Di solito sì, ma non sempre. Imporre a zero i bit di un intero fa sempre l'intero uguale a zero. Imporre a zero i bit di un numero a virgola mobile fa sempre il numero uguale a zero anche se questo non è garantito (dipende da come sono memorizzati i numeri a virgola mobile). La stessa cosa vale per i puntatori: un puntatore i cui bit sono uguali a zero non è necessariamente un puntatore nullo.

**\*D: Abbiamo capito che il meccanismo dei tag di struttura permette di creare una struttura di contenere un puntatore a se stessa. Ma cosa accade se le due strutture hanno un membro attraverso il quale punta l'unica struttura? [p. 439]**

R: Ecco come gestire questa situazione:

```
struct s1; /* dichiarazione incompleta di s1 */
struct s2 {
 struct s1 *p;
};
struct s1 {
 struct s2 *q;
};
```

La prima dichiarazione di s1 crea un tipo struttura incompleto (**tipi incompleti**) dato che non abbiamo specificato i membri di s1. La seconda dichiarazione è “completa” il tipo descrivendo i membri della struttura. Le dichiarazioni di una struttura sono permesse nel C sebbene il loro utilizzo sia limitato all'utilizzo di un puntatore a questo tipo (così come abbiamo fatto quando abbiamo dichiarato i puntatori a struttura).

**D: Chiamando la funzione malloc con un argomento sbagliato (che questa allochi troppa memoria o troppa poca) sembra essere un errore comune. C'è un modo più sicuro di utilizzare la funzione malloc?**

R: Sì, c'è. Alcuni programmatore seguono il seguente idioma quando chiamano la funzione malloc per allocare della memoria per un singolo oggetto:

```
p = malloc(sizeof(*p));
```

Poiché sizeof(\*p) è la dimensione dell'oggetto al quale punterà p, questa scrittura garantisce che venga allocata la quantità corretta di memoria. A prima vista questo idioma sembra sospetto: è probabile che p non sia inizializzata, il che rende \*p indefinito. Tuttavia sizeof non valuta \*p, ma calcola solamente la sua dimensione.

azione venga

ando i suoi  
nno uguali

ore diventare  
ibile rende il  
me vengono  
ri: un punta-  
llo.

permette a  
succede se  
na all'altra?

pletivi > 19.3]  
azione di si  
i incomplete  
o. Creare un  
arato p) cor-

(facendo si  
e un errore  
? [p. 440]  
chiamano la

ta istruzione  
vista questo  
e il valore di  
imensione e

quindi l'idioma funziona anche se p non è stata inizializzata o contiene un puntatore nullo.

Per allocare della memoria per un vettore con n elementi, possiamo utilizzare una versione leggermente modificata dell'idioma:

```
p = malloc(n * sizeof(*p));
```

**D: Perché la funzione qsort non è stata chiamata semplicemente sort? [p. 455]**

R: Il nome qsort deriva dall'algoritmo Quicksort pubblicato da C.A.R. Hoare nel 1962 (e discusso nella Sezione 9.6). Ironicamente lo standard C non richiede che qsort utilizzi l'algoritmo Quicksort, sebbene molte versioni di qsort lo facciano.

**D: È obbligatorio fare il cast al tipo void \* del primo argomento qsort, come accade nell'esempio seguente? [p. 455]**

```
qsort((void *) inventory, num_parts, sizeof(struct part),
 compare_parts);
```

R: No. Un puntatore a qualsiasi tipo può essere convertito automaticamente al tipo void \*.

**\*D: Vorremmo utilizzare qsort per ordinare un vettore di interi, ma stiamo incontrando dei problemi nello scrivere una funzione di confronto. Qual è il segreto?**

R: Ecco una versione che funziona:

```
int compare_ints(const void *p, const void *q)
{
 return *(int *)p - *(int *)q;
}
```

Bizzarro vero? L'espressione (int \*)p effettua il casting di p al tipo int \*. Quindi \*(int \*)p sarà l'intero al quale punta p. Un avvertimento: sottrarre due interi può causare un overflow. Se gli interi che vengono ordinati sono completamente arbitrari, è più sicuro utilizzare delle istruzioni if per confrontare \*(int \*)p con \*(int \*)q.

**\*D: Dovevamo ordinare un vettore di stringhe e quindi abbiamo pensato di utilizzare la funzione strcmp come funzione di confronto. Tuttavia, quando lo passiamo alla funzione qsort, il compilatore genera un warning. Abbiamo provato a risolvere il problema incorporando la funzione strcmp in una funzione di confronto:**

```
int compare_strings(const void *p, const void *q)
{
 return strcmp(p, q);
}
```

**Ora il nostro programma compila ma la qsort non sembra ordinare il vettore. Cosa stiamo sbagliando?**

R: Per prima cosa non potete passare la strcmp alla qsort dato che quest'ultima richiede una funzione di confronto con due parametri const void \*. La vostra funzione

`compare_strings` non funziona perché assume erroneamente che `p` e `q` siano stringhe (puntatori `char *`). Infatti `p` e `q` puntano a degli elementi di vettore contenenti dei puntatori `char *`. Per aggiustare la funzione `compare_strings` dobbiamo effettuare il casting di `p` e `q` al tipo `char **` e poi usare l'operatore `*` per rimuovere un livello di indirizzamento:

```
int compare_strings(const void *p, const void *q)
{
 return strcmp(*(char **)p, *(char **)q);
}
```

## Esercizi

- Sezione 17.1** 1. Dovet controllare ogni volta il valore restituito dalla funzione `malloc` (o quello di ogni funzione per l'allocazione della memoria) può essere scomodo. Scrivete una funzione chiamata `my_malloc` che serva da "wrapper" per la funzione `malloc`. Quando `my_malloc` viene invocata chiedendole di allocare `n` byte, questa chiama a sua volta la funzione `malloc` e si assicura che il valore restituito da quest'ultima non sia un puntatore nullo. La funzione restituirà il puntatore ottenuto dalla funzione `malloc`. Fate in modo che `my_malloc` stampi un messaggio di errore e termini il programma nel caso in cui `malloc` restituisse un puntatore nullo.
- Sezione 17.2** 2. Scrivete una funzione chiamata `duplicate` che utilizzi l'allocazione dinamica della memoria per creare una copia di una stringa. Per esempio, la chiamata  
 `p = duplicate(str);`  
allocherà dello spazio per una stringa della stessa lunghezza di `str`, copierà il contenuto di `str` nella nuova stringa e poi restituirà un puntatore a quest'ultima. La funzione dovrà restituire un puntatore nullo nel caso in cui l'allocazione della memoria non andasse a buon fine.
- Sezione 17.3** 3. Scrivete la seguente funzione:  
`int *create_array(int n, int initial_value);`  
La funzione dovrà restituire un puntatore a un vettore di `int` allocato dinamicamente e costituito da `n` elementi. Ogni elemento dovrà essere inizializzato al valore `initial_value`. Il valore restituito dovrà essere uguale a `NULL` nel caso in cui il vettore non possa essere allocato.

- Sezione 17.5** 4. Supponete che siano state effettuate le seguenti dichiarazioni:

```
struct point { int x, y; };
struct rectangle { struct point upper_left, lower_right; };
struct rectangle *p;
```

Vogliamo che il puntatore `p` punti a una struttura `rectangle` il cui vertice superiore sinistro si trovi nel punto (10, 25) mentre il vertice inferiore destro si trovi nel punto (20, 15). Scrivete una serie di istruzioni che allochino una struttura di questo tipo e che la inizializzi come indicato.

- W 5. Supponete che f e p siano dichiarate in questo modo:

```
struct {
 union {
 char a, b;
 int c;
 } d;
 int e[5];
} f, *p = &f;
```

Quali delle seguenti istruzioni sono corrette?

- (a) p->b = ' ';
- (b) p->e[3] = 10;
- (c) (\*p).d.a = '\*' ;
- (d) p->d->c = 20;

6. Modificate la funzione `delete_from_list` in modo che usi solamente una variabile puntatore invece di due (`cur` e `prev`).
- W 7. Il ciclo seguente è stato pensato per eliminare tutti i nodi di una lista concatenata e rilasciare la memoria occupata da questi. Sfortunatamente il ciclo non è corretto. Spiegate qual è il problema e mostrate come risolverlo.
- ```
for (p = first; p != NULL; p = p->next)
    free(p);
```
- W 8. La Sezione 15.2 descrive un file (`stack.c`) che fornisce delle funzioni che servono a salvare degli interi all'interno di uno stack. In quella sezione lo stack è stato implementato come un vettore. Modificate `stack.c` in modo che lo stack sia contenuto in una lista concatenata. Sostituite le variabili `contents` e `top` con una singola variabile che punti al primo nodo della lista (la "cima" dello stack). Scrivete le funzioni presenti in `stack.c` in modo che utilizzino dei puntatori. Rimuovete la funzione `is_full` e fate in modo che la funzione `push` restituisca il valore `true` se c'è memoria disponibile per creare il nodo, altrimenti restituisca il valore `false`.
9. Vero o falso: Se `x` è una struttura e `a` è un membro di quella struttura, allora `(&x)->a` è equivalente a `x.a`. Giustificate la vostra risposta.
10. Modificate la funzione `print_part` della Sezione 16.2 in modo che il suo parametro sia un puntatore a una struttura `part`. Nella vostra risposta utilizzate l'operatore `->`.

11. Scrivete la seguente funzione

```
int count_occurrences(struct node *list, int n);
```

Il parametro `list` punta a una lista concatenata, la funzione deve restituire il numero di volte in cui `n` compare nella lista. Assumete che la struttura `node` sia quella definita nella Sezione 17.5.

12. Scrivete la seguente funzione:

```
struct node *find_last(struct node *list, int n);
```

Il parametro `list` punta a una lista concatenata. La funzione deve restituire un puntatore all'ultimo nodo contenente `n`. Deve restituire `NULL` se `n` non compare nella lista. Assumete che la struttura `node` sia quella definita nella Sezione 17.5.

13. La funzione seguente è stata pensata per inserire un nuovo nodo nel punto appropriato all'interno di una lista ordinata. La funzione è stata pensata per restituire un puntatore al primo nodo della lista modificata. Sfortunatamente la funzione non agisce nel modo appropriato in tutti i casi che si possono presentare. Spieghate qual è il problema e mostrate come può essere risolto. Assumete che la struttura `node` sia definita nella Sezione 17.5.

```
struct node *insert_into_ordered_list(struct node *list,
                                      struct node *new_node)
{
    struct node *cur = list, *prev = NULL;
    while (cur->value <= new_node->value) {
        prev = cur;
        cur = cur->next;
    }
    prev->next = new_node;
    new_node->next = cur;
    return list;
}
```

- Sezione 17.6** 14. Modificate la funzione `delete_from_list` (Sezione 17.5) in modo che il suo primo parametro sia di tipo `struct node **` (un puntatore a un puntatore al primo nodo della lista) e il suo tipo restituito sia `void`. La funzione deve modificare il suo primo argomento in modo che punti alla lista dopo l'eliminazione del nodo desiderato.

- Sezione 17.7** 15. Mostrate l'output prodotto dal seguente programma e spiegate cosa fa.

```
#include <stdio.h>

int f1(int (*f)(int));
int f2(int i);

int main(void)
{
    printf("Answer: %d\n", f1(f2));
    return 0;
}
int f1(int (*f)(int))
{
    int n = 0;

    while ((*f)(n)) n++;
    return n;
}
int f2(int i)
{
    return i * i + i - 12;
}
```

16. Scrivete la funzione seguente. La chiamata `sum(g, i, j)` deve restituire `g(i) + ... + g(j)`.

```
int sum(int (*f)(int), int start, int end);
```

17. Sia `a` un vettore di 100 interi. Scrivete una chiamata alla funzione `qsort` che ordini solo gli ultimi 50 elementi del vettore `a` (non avete bisogno di scrivere la funzione di confronto).
18. Modificate la funzione `compare_parts` in modo che i componenti siano ordinati in ordine decrescente rispetto al numero di componente.
19. Scrivete una funzione che, quando le viene data una stringa per argomento, vada alla ricerca del nome di un comando corrispondente all'interno del seguente vettore di strutture. La funzione dovrà poi chiamare la funzione associata a quel nome.

```
struct {  
    char *cmd_name;  
    void (*cmd_pointer)(void);  
} file_cmd[] =  
{ {"new", new_cmd},  
  {"open", open_cmd},  
  {"close", close_cmd},  
  {"close all", close_all_cmd},  
  {"save", save_cmd},  
  {"save as", save_as_cmd},  
  {"save all", save_all_cmd},  
  {"print", print_cmd},  
  {"exit", exit_cmd}  
};
```

Progetti di programmazione

1. Modificate il programma `inventory.c` della Sezione 16.3 in modo che il vettore `inventory` venga allocato dinamicamente e successivamente riallocato al suo riempimento. Inizialmente utilizzate la funzione `malloc` per allocare lo spazio sufficiente per un vettore di 10 strutture `part`. Quando il vettore non ha più spazio per contenere nuovi componenti, utilizzate la funzione `realloc` per raddoppiare la sua dimensione. Ripetete il processo di raddoppio ogni volta che il vettore si riempie.
2. Modificate il programma `inventory.c` della Sezione 16.3 in modo che il comando `p` (print) chiama la funzione `qsort` per ordinare il vettore `inventory` prima di stampare l'elenco dei componenti.
3. Modificare il programma `inventory2.c` della Sezione 17.5 aggiungendogli il comando `e` (erase) che permette all'utente di rimuovere un componente dal database.

Il parametro `list` punta a una lista concatenata. La funzione deve restituire un puntatore all'ultimo nodo contenente `n`. Deve restituire `NULL` se `n` non compare nella lista. Assumete che la struttura `node` sia quella definita nella Sezione 17.5.

13. La funzione seguente è stata pensata per inserire un nuovo nodo nel punto appropriato all'interno di una lista ordinata. La funzione è stata pensata per restituire un puntatore al primo nodo della lista modificata. Sfortunatamente la funzione non agisce nel modo appropriato in tutti i casi che si possono presentare. Spiegate qual è il problema e mostrate come può essere risolto. Assumete che la struttura `node` sia definita nella Sezione 17.5.

```
struct node *insert_into_ordered_list(struct node *list,
                                      struct node *new_node)
{
    struct node *cur = list, *prev = NULL;
    while (cur->value <= new_node->value) {
        prev = cur;
        cur = cur->next;
    }
    prev->next = new_node;
    new_node->next = cur;
    return list;
}
```

- Sezione 17.6** 14. Modificate la funzione `delete_from_list` (Sezione 17.5) in modo che il suo primo parametro sia di tipo `struct node **` (un puntatore a un puntatore al primo nodo della lista) e il suo tipo restituito sia `void`. La funzione deve modificare il suo primo argomento in modo che punti alla lista dopo l'eliminazione del nodo desiderato.

- Sezione 17.7** 15. Mostrate l'output prodotto dal seguente programma e spiegate cosa fa.

```
#include <stdio.h>

int f1(int (*f)(int));
int f2(int i);

int main(void)
{
    printf("Answer: %d\n", f1(f2));
    return 0;
}
int f1(int (*f)(int))
{
    int n = 0;

    while ((*f)(n)) n++;
    return n;
}
int f2(int i)
{
    return i * i + i - 12;
}
```

16. Scrivete la funzione seguente. La chiamata `sum(g, i, j)` deve restituire `g(i) + ... + g(j)`.

```
int sum(int (*f)(int), int start, int end);
```

- W 17. Sia `a` un vettore di 100 interi. Scrivete una chiamata alla funzione `qsort` che ordini solo gli ultimi 50 elementi del vettore `a` (non avete bisogno di scrivere la funzione di confronto).

18. Modificate la funzione `compare_parts` in modo che i componenti siano ordinati in ordine decrescente rispetto al numero di componente.

19. Scrivete una funzione che, quando le viene data una stringa per argomento, vada alla ricerca del nome di un comando corrispondente all'interno del seguente vettore di strutture. La funzione dovrà poi chiamare la funzione associata a quel nome.

```
struct {  
    char *cmd_name;  
    void (*cmd_pointer)(void);  
} file_cmd[] =  
{ {"new", new_cmd},  
  {"open", open_cmd},  
  {"close", close_cmd},  
  {"close all", close_all_cmd},  
  {"save", save_cmd},  
  {"save as", save_as_cmd},  
  {"save all", save_all_cmd},  
  {"print", print_cmd},  
  {"exit", exit_cmd}  
};
```

Progetti di programmazione

- W 1. Modificate il programma `inventory.c` della Sezione 16.3 in modo che il vettore `inventory` venga allocato dinamicamente e successivamente riallocato al suo riempimento. Inizialmente utilizzate la funzione `malloc` per allocare lo spazio sufficiente per un vettore di 10 strutture `part`. Quando il vettore non ha più spazio per contenere nuovi componenti, utilizzate la funzione `realloc` per raddoppiare la sua dimensione. Ripetete il processo di raddoppio ogni volta che il vettore si riempie.
- W 2. Modificate il programma `inventory.c` della Sezione 16.3 in modo che il comando `p` (print) chiami la funzione `qsort` per ordinare il vettore `inventory` prima di stampare l'elenco dei componenti.
3. Modificare il programma `inventory2.c` della Sezione 17.5 aggiungendogli il comando `e` (erase) che permette all'utente di rimuovere un componente dal database.

4. Modificate il programma `justify.c` della Sezione 15.3 riscrivendo il file `line.c` in modo che salvi la riga corrente in una lista concatenata. Ogni nodo presente nella lista dovrà contenere una singola parola. Il vettore `line` dovrà essere sostituito da una variabile che punta al nodo contenente la prima parola. Questa variabile dovrà contenere un puntatore nullo nel caso in cui la riga fosse vuota.
5. Scrivete un programma che ordini una serie di parole immesse dall'utente:

```
Enter word: foo  
Enter word: bar  
Enter word: baz  
Enter word: quux  
Enter word:
```

In sorted order: bar baz foo quux

Assumete che ogni parola non sia più lunga di 20 caratteri. Interrompete la lettura quando l'utente immette una parola vuota (cioè pigia il tasto Invio senza immettere una parola). Salvate ogni parola in una stringa allocata dinamicamente utilizzando un vettore di puntatori per tenere traccia delle stringhe come nel programma `remind2.c` (Sezione 17.2). Dopo che tutte le parole sono state lette, ordinate il vettore (utilizzando una qualsiasi tecnica di ordinamento) e poi utilizzate un ciclo per far sì che stampi le parole in modo ordinato. Suggerimento: per leggere le parole utilizzate la funzione `read_line`, così com'è stato fatto nel programma `remind2.c`.

6. Modificate il Progetto di programmazione 5 in modo da utilizzare la `qsort` per ordinare il vettore di puntatori.
7. (C99)Modificate il programma `remind2.c` della Sezione 17.2 in modo che ogni elemento del vettore `reminders` sia un puntatore a una struttura `vstring` (guardate la Sezione 17.9) invece che un puntatore a una stringa ordinaria.

18 Dichiarazioni

Le dichiarazioni giocano un ruolo centrale nella programmazione C. Dichiarando le variabili e le funzioni forniamo delle informazioni vitali di cui il computer ha bisogno per controllare i potenziali errori di un programma e tradurre questo in codice oggetto.

I capitoli precedenti forniscono esempi di dichiarazioni senza entrare nel dettaglio, questo capitolo colma i vuoti. Esploreremo le sofisticate opzioni che possono essere utilizzate nelle dichiarazioni e vedremo che le dichiarazioni di variabili e funzioni hanno diverse cose in comune. Il capitolo fornirà inoltre una solida base per i concetti importanti della durata della memorizzazione, dello scope e del linking.

La Sezione 18.1 esamina la sintassi delle dichiarazioni nella loro forma più generale, un argomento che è stato evitato fino a questo momento. Le quattro sezioni successive si focalizzano sugli oggetti che compaiono nelle dichiarazioni: le classi di memorizzazione (Sezione 18.4) e gli inizializzatori (Sezione 18.5). La Sezione 18.6 tratta la keyword `inline` che può comparire nelle dichiarazioni di funzioni del C99.

18.1 Sintassi delle dichiarazioni

Le dichiarazioni forniscono al compilatore informazioni riguardanti il significato degli identificatori. Quando scriviamo

```
int i;
```

stiamo informando il compilatore che, nello scope corrente, il nome `i` rappresenta una variabile di tipo `int`. La dichiarazione

```
float f(float);
```

dice al compilatore che `f` è una funzione che restituisce un valore `float` e che possiede un argomento, anch'esso di tipo `float`.

In generale, una dichiarazione ha il seguente aspetto:

specificatori-di-dichiarazione dichiaratori

Gli **specificatori di dichiarazione** (*declaration specifiers*) descrivono le proprietà delle variabili o delle funzioni che sono state dichiarate. I **dichiaratori** (*declarators*) assegnano loro dei nomi e possono fornire delle informazioni aggiuntive sulle loro proprietà.

Gli specificatori di dichiarazione ricadono all'interno di tre categorie:

- **Classe di memorizzazione.** Vi sono quattro classi di memorizzazione: `auto`, `static`, `extern` e `register`. In una dichiarazione può comparire al massimo una classe di memorizzazione e, se presente, deve comparire come primo specificatore.
- **Qualificatori di tipo.** Nel C89 ci sono solamente due qualificatori di tipo: `const` e `volatile`. Il C99 possiede un terzo tipo di qualificatore: `restrict`. Una dichiarazione può contenere zero o più qualificatori di tipo.
- **Specificatori di tipo.** Le keyword `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed` e `unsigned` sono tutte specificatori di tipo. Queste parole possono essere combinate come descritto nel Capitolo 7. L'ordine nel quale compaiono non ha importanza (`int unsigned long` equivale a `long unsigned int`). Gli specificatori di tipo includono anche le specifiche delle strutture, delle unioni e delle enumerazioni (per esempio `struct point {int x, y;}`, `struct {int x, y;} o struct point`). Allo stesso modo anche i nomi creati utilizzando `typedef` sono specificatori.



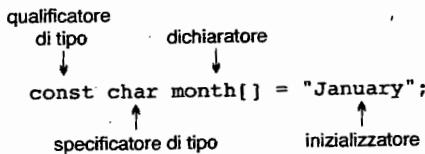
Il C99 possiede un quarto tipo di specificatore di dichiarazione: lo specificatore di funzione (*function specifier*) che viene utilizzato solamente nelle dichiarazioni delle funzioni. Questa categoria ha solamente un membro, la keyword `inline`. I qualificatori di tipo e gli specificatori di tipo devono seguire la classe di memorizzazione ma non possiedono altre restrizioni sull'ordine nel quale vengono inseriti. Per una questione di stile in questo libro porremo sempre i qualificatori prima degli specificatori di tipo.

Le dichiarazioni includono: identificatori (nomi di semplici variabili), identificatori seguiti da `[]` (nomi di vettori), identificatori preceduti da `*` (nomi di puntatori) e identificatori seguiti da `()` (nomi di funzione). Le dichiarazioni sono separate da virgolette. Un dichiaratore che rappresenti una variabile può essere seguito da un inizializzatore.

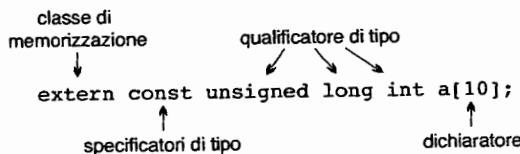
Guardiamo un paio di esempi che illustrano queste regole. Ecco una dichiarazione con classe di memorizzazione e tre dichiaratori:



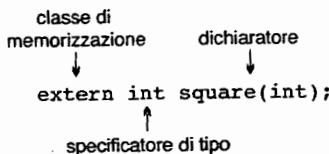
La dichiarazione seguente ha un qualificatore di tipo e un inizializzatore ma è priva di classe di memorizzazione:



La seguente dichiarazione ha sia la classe di memorizzazione che il qualificatore di tipo, inoltre presenta tre specificatori di tipo (il loro ordine non ha importanza):



Le dichiarazioni delle funzioni, come quelle delle variabili, possono avere una classe di memorizzazione, dei qualificatori di tipo e degli specificatori di tipo. La dichiarazione seguente possiede la classe di memorizzazione e uno specificatore di tipo:



Le prossime quattro sezioni trattano nel dettaglio le classi di memorizzazione, i qualificatori di tipo, i dichiaratori e gli inizializzatori.

18.2 Classi di memorizzazione

Le classi di memorizzazione possono essere specificate per le variabili e, in minore estensione, per le funzioni e i parametri. Per ora ci concentreremo sulle variabili.

Ricordiamo dalla Sezione 10.3 che il termine blocco si riferisce al corpo di una funzione (la parte racchiusa tra parentesi graffe) o a un'istruzione composta, che possa contenere delle dichiarazioni. Nel C99 le istruzioni di selezione (if e switch) e quelle di iterazione (while, do e for), assieme alle istruzioni "interne" che queste controllano, sono considerate a loro volta dei blocchi, sebbene questo sia praticamente un tecnicismo.

Proprietà delle variabili

Ogni variabile di un programma C presenta tre proprietà:

- **Durata di memorizzazione.** La durata di memorizzazione di una variabile determina quando la memoria viene riservata per la variabile e quando viene rilasciata. Lo spazio per una variabile con **durata di memorizzazione aut-**

D&R

tomatica viene allocato quando il blocco circostante viene eseguito. Lo spazio viene deallocato quando il blocco ha termine provocando così la perdita del valore posseduto dalla variabile. Una variabile con durata di memorizzazione statica permane nella stessa locazione di memoria per tutta la durata del programma, permettendo così che il suo valore venga mantenuto indefinitamente.

- **Scope.** Lo scope di una variabile è quella porzione del testo del programma all'interno della quale si può fare riferimento alla variabile stessa. Una variabile può avere sia **scope di blocco** (la variabile è visibile dal punto della sua dichiarazione fino alla fine del blocco) che **scope di file** (la variabile è visibile dal punto della sua dichiarazione fino alla fine del file che la contiene).

D&R

- **Collegamento.** Il collegamento (linkage) di una variabile determina l'estensione nella quale questa possa essere condivisa tra diverse parti del programma. Una variabile con **collegamento esterno** (*external linkage*) può essere condivisa tra diversi (anche tutti) i file di un programma. Una variabile con **collegamento interno** (*internal linkage*) è ristretta a un singolo file, ma può essere condivisa da tutte le funzioni presenti in quel file (se una variabile con lo stesso nome compare in un altro file, viene trattata come una variabile diversa). Una variabile **senza collegamento** (*no linkage*) appartiene a una singola funzione e non può essere condivisa.

Le proprietà di default per la durata di memorizzazione, lo scope e il collegamento di una variabile dipendono da dove questa viene dichiarata:

- le variabili dichiarate *dentro* un blocco (incluso il corpo di una funzione) hanno una durata di memorizzazione automatica, scope di blocco e sono senza collegamento;
- le variabili dichiarate *fuori* da qualsiasi blocco, nel livello più esterno di un programma hanno una durata di memorizzazione statica, scope di file e collegamento esterno.

L'esempio seguente illustra le proprietà di default per le variabili i e j:

```

int i;           // durata di memorizzazione statica
                // scope di file
                // collegamento esterno

void f(void)
{
    int j;           // durata di memorizzazione automatica
                    // scope di blocco
                    // priva di collegamento
}

```

Per molte variabili, le proprietà di default della durata di memorizzazione, dello scope e del collegamento sono adeguate. Quando non lo sono possiamo alterare queste proprietà specificando una classe di memorizzazione: auto, static, extern o register.

Classe di memorizzazione auto

La classe di memorizzazione auto è ammissibile solo per le variabili che appartengono a un blocco. Una variabile auto possiede una durata di memorizzazione automatica (non sorprendentemente), scope di blocco ed è senza collegamento. La classe di memorizzazione auto non viene specificata quasi mai esplicitamente perché è la situazione di default per le variabili dichiarate all'interno di un blocco.

Classe di memorizzazione static

La classe di memorizzazione static può essere utilizzata con tutte le variabili, indipendentemente da dove queste siano state dichiarate. Tuttavia ha un effetto diverso se applicata alle variabili dichiarate al di fuori di un blocco o alle variabili dichiarate all'interno di un blocco. Quando viene utilizzata *fuori* da un blocco, la parola static specifica che la variabile ha collegamento interno. Quando viene utilizzata *dentro* un blocco, static modifica la durata di memorizzazione da automatica a statica. La figura seguente illustra l'effetto della dichiarazione come static delle variabili i e j:

```
static int i; // durata di memorizzazione statica
              // scope di file
              // collegamento interno

void f(void)
{
    static int j; // durata di memorizzazione statica
                  // scope di blocco
                  // priva di collegamento
}
```

Quando viene utilizzata in una dichiarazione al di fuori di un blocco, la keyword static essenzialmente nasconde una variabile all'interno del file nel quale è stata dichiarata. Solo le funzioni che compaiono nello stesso file possono vedere la variabile. Nell'esempio seguente, le funzioni f1 e f2 hanno entrambe accesso alla variabile i, mentre le funzioni appartenenti ad altri file ne sono prive:

```
static int i;

void f1(void)
{
    /* ha accesso a i */
}

void f2(void)
{
    /* ha accesso a i */
}
```

Quest'uso della parola static aiuta a implementare una tecnica conosciuta come *information hiding* [information hiding > 19.2].

Una variabile statica dichiarata dentro un blocco risiede nella stessa locazione di memoria durante tutta l'esecuzione del programma. A differenza delle variabili auto-

matiche che perdono il loro valore ogni volta che il programma lascia il blocco che le contiene, una variabile statica manterrà il suo valore indefinitamente. Le variabili statiche possiedono alcune proprietà interessanti.

- Una variabile statica presente in un blocco viene inizializzata solamente una volta, ovvero prima dell'esecuzione del programma. Una variabile auto viene inizializzata ogni volta che viene a esistere (ammesso che abbia un inizializzatore, naturalmente).
- Ogni volta che una funzione viene chiamata ricorsivamente ottiene un nuovo insieme di variabili automatiche. Tuttavia, se possiede una variabile static, questa viene condivisa da tutte le chiamate alla funzione.
- Sebbene una funzione non debba restituire un puntatore a una variabile automatica, non c'è nulla di sbagliato nel restituire un puntatore a una variabile statica.

Dichiarare una delle sue variabili come statica permette a una funzione di mantenere delle informazioni tra le chiamate in un'area "nascosta" alla quale il resto del programma non può accedere. Tuttavia utilizzeremo più spesso la keyword static per rendere i programmi più efficienti. Considerate la funzione seguente:

```
char digit_to_hex_char(int digit)
{
    const char hex_chars[16] = "0123456789ABCDEF";
    return hex_chars[digit];
}
```

Ogni volta che la funzione `digit_to_hex_char` viene invocata, i caratteri 0123456789ABCDEF vengono copiati all'interno del vettore `hex_chars` per inizializzarlo. Ora rendiamo il vettore statico:

```
char digit_to_hex_char(int digit)
{
    static const char hex_chars[16] = "0123456789ABCDEF";
    return hex_chars[digit];
}
```

Dato che le variabili statiche vengono inizializzate solamente una volta, abbiamo incrementato la velocità della funzione.

Classe di memorizzazione extern

La classe di memorizzazione `extern` permette di condividere la stessa variabile tra diversi file sorgente. La Sezione 15.2 ha trattato i concetti fondamentali dell'utilizzo della keyword `extern`, per questo motivo non ci dilungheremo molto in questa sezione. Ricordiamo che la dichiarazione

```
extern int i;
```

informa il compilatore che `i` è una variabile `int`, ma non comporta l'allocazione di memoria per contenerla. Nella terminologia C, questa dichiarazione non è una definizione di `i`, informa solamente il compilatore del fatto che abbiamo bisogno

di accedere a una variabile che è stata definita altrove (forse in un punto successivo dello stesso file o, come accade più spesso, in un altro file). Una variabile può possedere molte dichiarazioni all'interno di un programma, ma deve avere solamente una definizione.

Vi è un'eccezione alla regola: le dichiarazioni `extern` non sono delle definizioni di variabile. Una dichiarazione `extern` che inizializzi una variabile funge come definizione della variabile stessa. Per esempio, la dichiarazione

```
extern int i = 0;
```

è di fatto equivalente a

```
int i = 0;
```

Questa regola previene che più dichiarazioni `extern` inizializzino una variabile in modo diverso.

Una variabile presente in una dichiarazione `extern` ha una durata di memorizzazione statica. Lo scope della variabile dipende dalla posizione della dichiarazione. Se la dichiarazione è all'interno di un blocco, la variabile ha scope di blocco, altrimenti ha scope di file:

```
extern int i;           durata di memorizzazione statica
                        scope di file
                        collegamento ?
```

```
void f(void)
{
    extern int j;       durata di memorizzazione statica
                        scope di blocco
                        collegamento ?
```

```
}
```

Determinare il tipo di collegamento di una variabile esterna è un po' più difficile. Se la variabile è stata dichiarata precedentemente nel file come `static` (al di fuori di qualsiasi definizione di funzione), allora possiede un collegamento interno. Altrimenti (il caso normale) la variabile avrà collegamento esterno.

Classe di memorizzazione `register`

Utilizzare la classe di memorizzazione `register` nella dichiarazione di una variabile equivale a chiedere al compilatore di memorizzare questa in un registro invece di mantenerla nella memoria principale come avviene con le altre variabili. (Un registro è un'area di memoria collocata all'interno della CPU del computer. I dati contenuti in un registro sono accessibili e aggiornabili più velocemente rispetto a quelli contenuti nella memoria normale.) Specificare la classe di memorizzazione di una variabile come `register` è una richiesta e non un comando. Il compilatore, se lo vuole, è libero di memorizzare una variabile `register` nella memoria.

La classe di memorizzazione `register` è ammessa solo per le variabili dichiarate all'interno di un blocco. Una variabile `register` possiede la stessa durata di memorizzazione, scope e collegamento delle variabili automatiche. Tuttavia una variabile `register` differisce per una cosa dalle variabili automatiche: visto che i registri non

hanno un indirizzo non è possibile utilizzare l'operatore & per ottenerne l'indirizzo. Questa restrizione si applica anche se il compilatore ha deciso di memorizzare la variabile nella memoria.

La keyword register viene utilizzata soprattutto per le variabili utilizzate e/o ag-
giornate frequentemente. Per esempio, la variabile di controllo di un ciclo for è una
buona candidata per essere dichiarata come register:

```
int sum_array(int a[], int n)
{
    register int i;
    int sum = 0;

    for (i = 0; i < n; i++)
        sum += a[i];
    return sum;
}
```

Tra i programmatore C la keyword register non è più popolare come in passato. I
compilatori odierni sono molto più sofisticati dei primi compilatori C, molti infatti
possono determinare automaticamente quali variabili possano beneficiare dall'essere
 contenute all'interno di un registro. In ogni caso utilizzare questa keyword fornisce
 informazioni utili che possono aiutare il compilatore a migliorare le performance del
 programma. In particolare, il compilatore sa che non è possibile ottenere l'indirizzo
 di variabile register e quindi che non può essere modificata per mezzo di un punta-
 tore. Sotto questo aspetto la keyword register è imparentata con la keyword del C99
 restrict.

Classe di memorizzazione di una funzione

Le dichiarazioni (e le definizioni) delle funzioni, come le dichiarazioni delle variabili,
 possono includere una classe di memorizzazione, ma le uniche opzioni disponibili
 sono extern e static. La parola extern all'inizio della dichiarazione di una funzione
 specifica che la funzione ha un collegamento esterno, permettendo così che essa possa
 essere chiamata da altri file. La parola static indica un collegamento interno, limitan-
 do in questo modo l'uso del nome della funzione al file dove questa è definita. Se la
 classe di memorizzazione non viene specificata, viene assunto che la funzione abbia
 collegamento esterno.

Considerate le seguenti dichiarazioni di funzione:

```
extern int f(int i);
static int g(int i);
int h(int i);
```

f ha collegamento esterno, g ha collegamento interno mentre h (per default) ha col-
 legamento esterno. A causa del suo collegamento interno, g non può essere chiamata
 direttamente dall'esterno del file nel quale è dichiarata (dichiarare g come static non
 impedisce completamente che questa venga chiamata da un altro file: una chiamata
 indiretta attraverso un puntatore a funzione è ancora possibile).

Dichiarare funzioni come `extern` è come dichiarare le variabili `auto` (non ha scopo). Per questa ragione nel presente volume non utilizziamo la keyword `extern` nelle dichiarazione delle funzioni. Tuttavia siate consapevoli del fatto che molti programmati fanno un uso intensivo di questa keyword, il che certamente non crea danni.

Dichiarare una funzione come `static`, d'altro canto, è abbastanza utile. Infatti è raccomandabile l'uso della keyword `static` quando viene dichiarata una funzione che non è pensata per essere chiamata da altri file. I benefici di questa pratica includono:

- **manutenzione più semplice.** Dichiarare la funzione `f` come `static` garantisce che questa non sia visibile al di fuori del file nel quale compare la sua definizione. Qualcuno che dovesse modificare il programma in un secondo momento saprebbe che le modifiche apportate a `f` non hanno effetti sugli altri file (una eccezione: una funzione in un altro file che viene passata come puntatore a `f` potrebbe risentire delle modifiche a `f`). Fortunatamente questa situazione è facilmente individuabile esaminando il file nel quale viene definita `f` visto che al suo interno deve essere definita anche la funzione che passa `f`);
- **riduzione dell'“inquinamento dello spazio dei nomi”.** Dato che le funzioni dichiarate `static` possiedono un collegamento interno, i loro nomi possono essere riutilizzati in altri file. Sebbene non vorremo mai riutilizzare deliberatamente il nome di una funzione per altri scopi, questo potrebbe essere difficile da evitare in programmi di grandi dimensioni. Un numero eccessivo di nomi con collegamento esterno può provocare quello che i programmati C chiamano “inquinamento dello spazio dei nomi”: nomi presenti in file differenti che entrano accidentalmente in conflitto gli uni con gli altri. Utilizzare la keyword `static` aiuta a prevenire questo problema.

I parametri delle funzioni hanno le stesse proprietà delle variabili automatiche: duttata di memorizzazione automatica, scope di blocco e nessun collegamento. L'unica classe di memorizzazione che può essere specificata per i parametri è la `register`.

Riepilogo

Ora che abbiamo trattato le varie classi di memorizzazione, riassumiamo quanto appreso. Il seguente frammento di programma illustra tutti i possibili modi per includere (oppure omettere) la classe di memorizzazione nelle dichiarazioni di variabili e parametri.

```
int a;
extern int b;
static int c;

void f(int d, register int e)
{
    auto int g;
    int h;
    static int i;
    extern int j;
    register int k;
}
```

La Tabella 18.1 illustra le proprietà di ogni variabile e parametro dell'esempio.

Tabella 18.1 Proprietà di variabili e parametri

Nome	Durata di memorizzazione	Scope	Collegamento
a	statica	file	esterno
b	statica	file	†
c	statica	file	interno
d	automatica	blocco	nessuno
e	automatica	blocco	nessuno
g	automatica	blocco	nessuno
h	automatica	blocco	nessuno
i	statica	blocco	nessuno
j	statica	blocco	†
k	automatica	blocco	nessuno

† Le definizioni di b e j non sono state mostrate e quindi non è possibile determinare il tipo di collegamento di queste variabili. Nella maggior parte dei casi le variabili sono definite in un altro file o possiedono un collegamento esterno.

Delle quattro classi di memorizzazione, le più importanti sono quella static e quella extern. La classe auto non ha alcun effetto e i compilatori moderni hanno reso la classe register meno importante.

18.3 Qualificatori di tipo



Vii sono due qualificatori di tipo: const e volatile (il C99 possiede un terzo qualificatore chiamato restrict che viene utilizzato solo con i puntatori [puntatori restricted > 17.8]). Dato che l'uso di volatile è limitato solo alla programmazione a basso livello, rimandiamo la sua trattazione alla Sezione 20.3. La keyword const viene usata per dichiarare degli oggetti che sembrano delle variabili ma sono a "sola lettura": un programma può accedere al valore di un oggetto const ma non può modificarlo. Per esempio, la dichiarazione

```
const int n = 10;
```

crea un oggetto const chiamato n il cui valore è uguale a 10. La dichiarazione

```
const int tax_brackets[] = {750, 2250, 3750, 5250, 7000};
```

crea un vettore const chiamato tax_brackets.

Dichiarare un oggetto come const ha diversi vantaggi.

- È una forma di documentazione: avvisa tutti quelli che leggono il programma della natura di sola lettura dell'oggetto.
- Il compilatore può controllare che il programma non cerchi inavvertitamente di modificare il valore dell'oggetto.

- Quando i programmi vengono scritti per certi tipi di applicazioni (in particolare i sistemi embedded), il compilatore può usare la parola const per identificare i dati che sono memorizzati nella ROM (read-only memory).

A prima vista può sembrare che la keyword const attenda allo stesso ruolo della direttiva #define che abbiamo usato nei capitoli precedenti per creare dei nomi per le costanti. Tuttavia ci sono differenze significative tra #define e const.

- Possiamo usare #define per dare un nome a costanti numeriche, carattere o stringhe costanti. La keyword const può essere usata per creare degli oggetti a sola lettura di qualsiasi tipo, inclusi vettori, puntatori, strutture e unioni.
- Gli oggetti const sono soggetti alle stesse regole di scope delle variabili, mentre le costanti create utilizzando #define non lo sono. In particolare non possiamo usare #define per creare una costante con scope di blocco.
- Il valore di un oggetto const, a differenza del valore di una macro può essere analizzato in un debugger.
- A differenza delle macro, gli oggetti const non possono essere usati nelle espressioni costanti. Per esempio, non possiamo scrivere

```
const int n = 10;  
int a[n];      /** SBAGLIATO ***/.
```

perché i confini dei vettori devono essere delle espressioni costanti (nel C99 questo esempio sarebbe ammissibile se a avesse una durata di memorizzazione automatica, infatti verrebbe trattato come un vettore di lunghezza variabile, viceversa non sarebbe ammissibile se a avesse una durata di memorizzazione statica).

- Dato che possiede un indirizzo, a un oggetto const è possibile applicare l'operatore di indirizzo &(). Una macro non ha un indirizzo.

Non ci sono regole assolute che stabiliscano quando usare #define e quando usare const. L'uso di #define è raccomandabile per le costanti che rappresentano numeri o caratteri. In questo modo sarete in grado di utilizzare le costanti come dimensioni dei vettori, nelle istruzioni switch e in tutti quei punti dove sono richieste le espressioni costanti.

18.4 Dichiaratori

Un dichiaratore consiste di un identificatore (il nome di una variabile o una funzione che vengono dichiarate) che può essere preceduto dal simbolo * o seguito da [] o (). Combinando *, [], () possiamo creare dichiaratori complessi a piacere.

Prima di affrontare dichiaratori più complicati riepiloghiamo quanto abbiamo visto nei primi capitoli. Nel caso più semplice, un dichiaratore è costituito semplicemente da un identificatore, come nell'esempio seguente:

```
int i;
```

I dichiaratori possono contenere anche i simboli *, [] e () .

- Un dichiaratore che inizia con * rappresenta un puntatore:

```
int *p;
```

- Un dichiaratore che termina con [] rappresenta un vettore:

```
int a[10];
```

Le parentesi quadre possono essere lasciate vuote se il vettore è un parametro, se ha un inizializzatore o se la sua classe di memorizzazione è extern:

```
extern int a[];
```

C99

Dato che a è stata definita altrove, il compilatore non ha bisogno di conoscere la sua lunghezza in questo punto (nel caso di un vettore multidimensionale, solamente il primo set di parentesi può essere lasciato vuoto). Il C99 fornisce due opzioni aggiuntive per quello che può essere messo tra le parentesi nella dichiarazione di un parametro vettore. Un'opzione è la keyword static seguita da un'espressione che specifica la lunghezza minima del vettore. L'altra opzione è il simbolo * che può essere usato nel prototipo di una funzione per indicare un argomento costituito da un vettore a lunghezza variabile. La Sezione 9.3 tratta entrambe queste caratteristiche del C99.

- Un dichiaratore che termina con () rappresenta una funzione:

```
int abs(int i);
void swap(int *a, int *b);
int find_largest(int a[], int n);
```

Il C permette che nella dichiarazione di una funzione i nomi dei parametri vengano omessi:

```
int abs();
void swap();
int find_largest();
```

Le parentesi possono anche essere lasciate vuote:

```
int abs();
void swap();
int find_largest();
```

Le dichiarazioni presenti nell'ultimo gruppo specificano i valori restituiti dalle funzioni abs, swap e find_largest ma non forniscono alcuna informazione sui loro argomenti. Lasciare le parentesi vuote non equivale a mettere la parola void tra esse, il che indicherebbe che non ci sono argomenti. Lo stile con le dichiarazioni delle funzioni con le parentesi vuote è praticamente scomparso: è uno stile inferiore rispetto a quello dei prototipi introdotto nel C89, dato che non permette al compilatore di controllare se le chiamate a funzione hanno gli argomenti corretti.

Se tutti i dichiaratori fossero semplici come questi, la programmazione C sarebbe una cosa semplicissima. Sfortunatamente i dichiaratori dei programmi veri combinano spesso le notazioni *, [] e (). Abbiamo già visto esempi di questo tipo. Sappiamo che

```
int *ap[10];
```

è la dichiarazione di un vettore di 10 puntatori a intero. Sappiamo che con float *fp(float);

dichiariamo una funzione che ha un argomento float e restituisce un puntatore a un float. Inoltre nella Sezione 17.7 abbiamo imparato che

```
void (*pf)(int);
```

dichiara un puntatore a una funzione con un argomento int e tipo restituito void.

Decifrare dichiarazioni complesse

Fino a ora non abbiamo incontrato grandi problemi nella comprensione dei dichiaratori, ma cosa possiamo dire di dichiaratori come quello seguente?

```
int *(*x[10])(void);
```

Questo dichiaratore combina *, [] e () e quindi non è ovvio se x sia un puntatore, un vettore o una funzione.

Fortunatamente ci sono due semplici regole che ci permettono di comprendere qualsiasi dichiarazione, indipendentemente da quanto sia involuta.

- **Leggere sempre i dichiaratori dall'interno.** In altre parole, dobbiamo individuare l'identificatore che si sta dichiarando e iniziare a decifrare la dichiarazione da quel punto.
- **Quando bisogna scegliere, privilegiate sempre [] e () al posto di *.** Se l'identificatore viene preceduto dal simbolo * e seguito da [], allora rappresenta un vettore e non un puntatore. Analogamente se l'identificatore è preceduto da * e seguito da () vuol dire che rappresenta una funzione (naturalmente possiamo sempre usare delle parentesi per annullare la normale priorità di [] e () rispetto a *).

Applichiamo queste regole al nostro esempio. Nella dichiarazione

```
int *ap[10];
```

l'identificatore è ap. Dato che ap è preceduto da * e seguito da [], diamo precedenza a [] e quindi ap è un vettore di puntatori. Nella dichiarazione

```
float *fp(float);
```

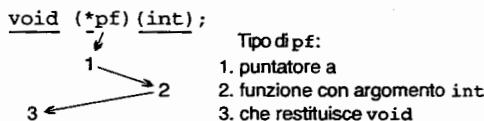
l'identificatore è fp. Visto che fp è preceduto da * ma seguito da (), diamo precedenza alle parentesi tonde e quindi fp è una funzione che restituisce un puntatore.

La dichiarazione

```
void (*pf)(int);
```

è leggermente complicata. Poiché la parte *pf è racchiusa tra parentesi, pf deve essere un puntatore. Tuttavia (*pf) è seguita da (int) e quindi pf deve puntare a una funzione con un argomento di tipo int. La parola void rappresenta il tipo restituito da questa funzione.

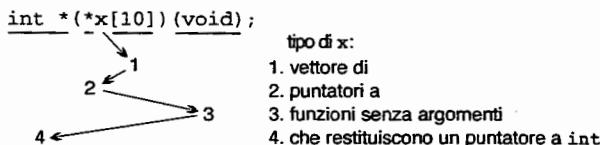
Come illustra l'ultimo esempio, comprendere un dichiaratore di tipo complesso spesso richiede di procedere a zigzag da un lato all'altro dell'identificatore:



Utilizziamo questa tecnica a zigzag per decifrare la dichiarazione fornita in precedenza:

```
int *(*x[10])(void);
```

Per prima cosa individuiamo l'identificatore oggetto della dichiarazione, ovvero x. Possiamo vedere che x è preceduto da * e seguito da []. Dato che le parentesi quadre hanno precedenza andiamo a destra (x è un vettore). Successivamente ci spostiamo a sinistra per capire il tipo degli elementi del vettore (puntatori). Successivamente torniamo a destra per capire a che tipo di dati facciano riferimento questi puntatori (funzioni senza argomenti). Infine andiamo a sinistra per capire cosa restituiscano queste funzioni (un puntatore a int). Ecco come si presenta graficamente il processo di decifrazione appena svolto:



Padroneggiare le dichiarazioni del C richiede tempo e pratica. L'unica buona notizia è che ci sono delle cose che non possono essere dichiarate in C. Le funzioni non possono restituire vettori:

```
int f(int[]); /** SBAGLIATO **/
```

Le funzioni non possono restituire funzioni:

```
int g(int)(int); /** SBAGLIATO **/
```

Non sono possibili nemmeno vettori di funzioni:

```
int a[10](int); /** SBAGLIATO **/
```

In ogni caso possiamo utilizzare i puntatori per ottenere l'effetto desiderato. Una funzione non può restituire un vettore ma può restituire un puntatore a un vettore. Una funzione non può restituire una funzione ma può restituire un puntatore a una funzione. I vettori di funzioni non sono permessi ma un vettore può contenere dei puntatori a funzione (la Sezione 17.7 ha fornito un esempio di questo tipo di vettori).

Usare le definizioni di tipo per semplificare le dichiarazioni

Alcuni programmati utilizzano le definizioni di tipo per semplificare le dichiarazioni più complesse. Considerate la dichiarazione di `x` che abbiamo esaminato precedentemente in questa sezione:

```
int *(*x[10])(void);
```

Per rendere il tipo di `x` più facilmente comprensibile, possiamo usare la seguente serie di definizioni di tipo:

```
typedef int *Fcn(void);
typedef Fcn *Fcn_ptr;
typedef Fcn_ptr Fcn_ptr_array[10];
Fcn_ptr_array x;
```

Se leggiamo queste righe in ordine inverso, vediamo che `x` è di tipo `Fcn_ptr_array`, che `Fcn_ptr_array` è un vettore di valori `Fcn_ptr`, che `Fcn_ptr` è un puntatore al tipo `Fcn` e che `Fcn` è una funzione priva di argomenti che restituisce un puntatore a un valore `int`.

18.5 Inizializzatori

Per ragioni di comodità il C ci permette di specificare i valori iniziali delle variabili al momento della loro dichiarazione. Per inizializzare una variabile scriviamo il simbolo `=` dopo il suo dichiaratore e poi lo facciamo seguire da un inizializzatore (non confondete il simbolo `=` presente in una dichiarazione con l'operatore di assegnamento. L'inizializzazione non equivale a un assegnamento).

Nei capitoli precedenti abbiamo visto vari tipi di inizializzatori. L'inizializzatore per una semplice variabile è un'espressione del tipo della variabile stessa:

```
int i = 5 / 2; /* i è inizialmente uguale a 2 */
```

Se i tipi non corrispondono, il C converte l'inizializzatore utilizzando le stesse regole usate negli assegnamenti [**conversioni durante gli assegnamenti > 7.4**]:

```
int j = 5.5; /* convertito in 5 */
```

L'inizializzatore per una variabile puntatore deve essere un'espressione puntatore dello stesso tipo della variabile o del tipo `void`:

```
int *p = &i;
```

Solitamente l'inizializzatore per un vettore, una struttura, o un'unione è costituito da una serie di valori racchiusi tra parentesi graffe:

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

Nel C99 gli inizializzatori racchiusi tra parentesi graffe possono seguire un altro formato grazie all'uso degli inizializzatori designati [**inizializzatori designati > 8.1, 16.1**]:

- Un inizializzatore per una variabile con durata di memorizzazione statica deve essere costante:

```
#define FIRST 1
#define LAST 100

static int i = LAST - FIRST + 1;
```

Visto che LAST e FIRST sono delle macro, il compilatore è in grado di calcolare il valore iniziale di i ($100 - 1 + 1 = 100$). Se LAST e FIRST fossero state variabili, l'inizializzatore non sarebbe stato ammissibile.

- Se una variabile ha una durata di memorizzazione automatica, il suo inizializzatore non deve essere necessariamente costante:

```
int f(int n)
{
    int last = n - 1;
}
```

- Un inizializzatore per un vettore, una struttura o un'unione racchiuso tra parentesi graffe deve contenere solamente un'espressione costante e mai variabili o chiamate a funzione:

```
#define N 2
int powers[5] = {1, N, N * N, N * N * N, N * N * N * N};
```

Dato che N è una costante, l'inizializzatore per il vettore powers è ammissibile. Se N fosse stata una variabile il programma non sarebbe stato compilabile. Nel C99 questa restrizione si applica solo se la variabile ha una durata di memorizzazione statica.

- Gli inizializzatori per le strutture o le unioni automatiche possono essere costituiti da un'altra struttura o unione:

```
void g(struct part part1)
{
    struct part part2 = part1;
}
```

L'inizializzatore non deve essere necessariamente una variabile o il nome di un parametro, sebbene necessiti di essere un'espressione del tipo appropriato. Per esempio, l'inizializzatore di part2 può essere `*p`, dove p è del tipo `struct part *`, oppure `f(part1)`, dove f è una funzione che restituisce una struttura part.

C99

Variabili non inizializzate

Nel capitoli precedenti abbiamo sottointeso che le variabili non inizializzate hanno dei valori indefiniti. Questo non è sempre vero. Il valore iniziale di una variabile dipende dalla sua durata di memorizzazione:

- le variabili con durata di memorizzazione automatica non hanno un valore iniziale di default. Il valore iniziale di una variabile automatica non può essere predefinito e può essere diverso ogni volta che la variabile viene a esistere;
- le variabili con durata di memorizzazione statica hanno per default il valore zero. A differenza della memoria allocata dalla funzione `calloc` [funzione `calloc` > 17.3], che semplicemente impone a zero i bit, una variabile statica viene inizializzata correttamente in base al suo tipo: le variabili intere vengono inizializzate a 0, le variabili a virgola mobile vengono inizializzate a 0.0 e le variabili puntatore vengono a contenere un puntatore nullo.

Per ragioni di stile è meglio fornire degli inizializzatori per le variabili statiche invece di basarsi sul fatto che c'è la garanzia che vengano impostate a zero. Se un programma accede a una variabile che non viene inizializzata esplicitamente, qualcuno che leggesse il programma non potrebbe determinare facilmente se la variabile è assunta uguale a zero o se viene inizializzata in un altro punto del programma.

18.6 Funzioni inline (C99)

Le dichiarazioni delle funzioni del C99 hanno un'opzione aggiuntiva che non esiste nel C89: possono contenere la keyword `inline`. Questa keyword è un nuovo tipo di specificatore di dichiarazione, che è distinta dalle classi di memorizzazione, dai qualificatori di tipo o dagli specificatori di tipo. Per capire l'effetto di una funzione `inline`, abbiamo bisogno di visualizzare le istruzioni macchina che vengono generate dal compilatore C per gestire il processo della chiamata a funzione e di ritorno dalla chiamata.

A livello macchina, in preparazione alla chiamata devono essere eseguite diverse istruzioni. La stessa chiamata richiede un salto alla prima istruzione della funzione, inoltre la funzione stessa può eseguire diverse istruzioni prima del suo avvio. Se la funzione possiede degli argomenti, questi hanno bisogno di essere copiati (a causa del fatto che il C passa i suoi argomenti per valore). Ritornare da una funzione richiede uno sforzo simile sia da parte della funzione che è stata chiamata che da quella che l'ha invocata. Il lavoro complessivo necessario per chiamare una funzione e ritornare da questa viene chiamato overhead perché rappresenta uno sforzo aggiuntivo oltre a quello necessario alla funzione per svolgere il compito per il quale è stata pensata. Sebbene l'overhead di una chiamata a funzione rallenti il programma solo in piccola parte, può aumentare in situazioni come quelle che si hanno quando una funzione viene chiamata milioni o miliardi di volte, quando si sta utilizzando un vecchio e lento processore (come nel caso dei sistemi embedded) o quando il programma deve rispettare scadenze molto stringenti (come nei sistemi real-time).

Nel C89 l'unico modo per ovviare all'overhead di una chiamata a funzione è quello di usare una macro parametrica [macro parametrica > 14.3]. Tuttavia le macro parametriche presentano alcuni inconvenienti. Il C99 offre una soluzione migliore a questo problema: creare una funzione `inline`. Il termine "inline" suggerisce una strategia di implementazione nella quale il compilatore rimpiazza ogni chiamata alla funzione con le istruzioni macchina della funzione stessa. Questa tecnica evita il normale overhead di una chiamata a funzione, sebbene possa provocare un incremento marginale della dimensione del programma compilato.

Dichiarare una funzione come `inline` non forza effettivamente il compilatore a rendere la funzione "inline". Semplicemente suggerisce che il compilatore dovrebbe provare a rendere le chiamate a quella funzione il più veloci possibile, magari eseguendo l'espansione `inline` quando la funzione viene chiamata. Il compilatore è libero di ignorare questi suggerimenti. Sotto questo aspetto la keyword `inline` è simile alle keyword `register` e `restrict` che possono essere usate dal compilatore per migliorare le performance del programma, ma possono anche essere ignorate.

Definizioni inline

Una funzione inline presenta la keyword `inline` come uno dei suoi specificatori di dichiarazione:

```
inline double average(double a, double b)
{
    return (a + b) / 2;
}
```

Qui le cose si fanno un po' più complicate. La funzione `average` ha collegamento esterno e quindi gli altri file sorgente possono contenere delle chiamate a questa funzione. Tuttavia la definizione di `average` non viene considerata dal compilatore come una definizione esterna (è una definizione inline) e quindi cercare di chiamare `average` da un altro file verrebbe considerato un errore.

Ci sono due modi per evitare questo errore. Una possibilità è quella di aggiungere la parola `static` nella definizione della funzione:

```
static inline double average(double a, double b)
{
    return (a + b) / 2;
}
```

Ora `average` ha collegamento interno e quindi non può essere chiamata da altri file. Gli altri file possono contenere una loro definizione di `average` che può essere uguale a questa definizione oppure essere diversa.

L'altra possibilità è di fornire una definizione esterna per `average` in modo che le chiamate vengano fatte anche da altri file. Un modo per farlo è quello di scrivere la funzione `average` una seconda volta (senza usare `inline`) e mettere la seconda definizione in un file sorgente diverso. Fare questo è ammissibile anche se non è una buona idea avere due versioni della stessa funzione perché non possiamo garantire che queste rimangano consistenti in caso di modifiche al programma.

Esiste un approccio migliore al problema. Per prima cosa inseriamo la definizione `inline` di `average` in una file header (chiamiamolo `average.h`):

```
#ifndef AVERAGE_H
#define AVERAGE_H

inline double average(double a, double b)
{
    return (a + b) / 2;
}

#endif
```

Successivamente creiamo un file sorgente corrispondente (`average.c`):

```
#include "average.h"

extern double average(double a, double b);
```

Adesso qualsiasi file che avesse bisogno di chiamare la funzione `average` dovrà semplicemente includere il file `average.h` che contiene la definizione `inline` della funzione. Il file `average.c` contiene un prototipo per la funzione che utilizza la keyword `extern`, la quale fa sì che la definizione inclusa da `average.h` venga trattata all'interno di `average.c` come una definizione esterna.

Una regola generale del C99 stabilisce che se tutte le dichiarazioni di livello più alto di una funzione presenti in un particolare file includono la keyword `inline` e non quella `extern`, allora la definizione della funzione presente in quel file è `inline`. Se la funzione è utilizzata altrove nel programma (incluso il file contenente la definizione `inline`), allora una definizione esterna della funzione deve essere fornita da qualche altro file. Quando la funzione viene chiamata, il compilatore può scegliere se eseguire una chiamata ordinaria (usando la definizione esterna della funzione) oppure eseguire l'espansione `inline` (usando la definizione `inline` della funzione). Non c'è modo di predire quale sarà la scelta intrapresa dal compilatore, per questo è vitale che le due definizioni siano consistenti. La tecnica che abbiamo appena discusso (usare i file `average.h` e `average.c`) garantisce che le definizioni siano uguali.

Restrizioni per le funzioni inline

Visto che le funzioni `inline` vengono implementate in un modo che è piuttosto diverso da quello delle funzioni ordinarie, sono soggette a regole differenti e a restrizioni. Le variabili con durata di memorizzazione statica sono particolarmente problematiche per le funzioni `inline` con collegamento esterno. Di conseguenza il C99 impone alle funzioni `inline` con collegamento esterno (ma non a quelle con collegamento interno) le seguenti restrizioni:

- la funzione non può definire una variabile `static` modificabile;
- la funzione non può contenere riferimenti a variabili con collegamento interno.

A una funzione di questo tipo è permesso definire una variabile che sia contemporaneamente `static` e `const`, tuttavia ogni definizione `inline` della funzione può creare una sua copia della variabile.

Usare le funzioni inline con GCC

Alcuni compilatori come GCC supportano le funzioni inline da prima dello standard C99. Ne risulta che le loro regole nell'uso delle funzioni inline possono differire dalle standard. In particolare, lo schema descritto precedentemente (usare i file `average.h` e `average.c`) potrebbe non funzionare con questi compilatori. Ci si aspetta che la versione 4.3 di GCC (non disponibile al momento della scrittura di questo libro) supporti le funzioni inline nel modo descritto dallo standard C99.

Le funzioni che sono specificate sia come `static` che come `inline` dovrebbero funzionare bene indipendentemente dalla versione di GCC. Questa strategia è ammessa.

anche in C99 e quindi è la scelta più sicura. Una funzione static inline può essere usata all'interno di un singolo file o messa in un file header e inclusa dentro tutti i file sorgente che hanno bisogno di chiamare la funzione.

C'è un altro modo per condividere una funzione inline tra diversi file che funziona con le vecchie versioni di GCC ma va in conflitto con lo standard C99. Questa tecnica richiede che la definizione della funzione venga messa in un file header, che la funzione venga specificata sia come extern che come inline, e che il file header venga incluso in tutti i file sorgente contenenti una chiamata alla funzione. Una seconda copia della definizione (senza le parole extern e inline) viene posta in uno dei file sorgente (in questo modo se per qualche motivo il compilatore non è in grado di rendere "inline" la funzione, questa ha comunque una definizione).

Un'osservazione finale a riguardo di GCC: le funzioni vengono rese "inline" solo quando viene richiesta l'ottimizzazione attraverso l'opzione -O della riga di comando.

Domande & Risposte



***D: Perché le istruzioni di selezione e quelle di iterazione (e le loro istruzioni "interne") vengono considerate come blocchi nel C99? [p. 475]**

R: Questa regola abbastanza sorprendente deriva da un problema che può verificarsi quando i letterali composti [**letterali composti > 9.3, 16.2**] vengono utilizzati nelle istruzioni di selezione e in quelle di iterazione. Il problema ha a che fare con la durata di memorizzazione dei letterali composti, quindi soffermiamoci un attimo a discutere questo argomento.

Lo standard C99 precisa che l'oggetto rappresentato da un letterale composto abbia una durata di memorizzazione statica se il letterale si trova fuori dal corpo di una funzione. In caso contrario ha una durata di memorizzazione automatica e ne risulta che la memoria occupata dall'oggetto viene deallocata alla fine del blocco nel quale compare il letterale composto stesso. Considerate la seguente funzione che restituisce una struttura point creata usando un letterale composto:

```
struct point create_point(int x, int y)
{
    return (struct point) {x, y};
}
```

Questa funzione si comporta correttamente perché l'oggetto creato dal letterale composto viene copiato quando la funzione ha termine. L'oggetto originale non esiste più ma la copia permane. Supponete ora di modificare leggermente la funzione:

```
struct point *create_point(int x, int y)
{
    return &(struct point) {x, y};
}
```

Questa versione di create_point è affetta da un comportamento indefinito perché restituisce un puntatore a un oggetto che ha durata di memorizzazione automatica e quindi cesserà di esistere quando la funzione avrà termine.

Ritorniamo ora alla domanda con la quale abbiamo cominciato: perché le istruzioni di selezione e di iterazione vengono considerate blocchi? Considerate l'esempio seguente:

```
/* Esempio 1 - istruzione if senza parentesi graffe */

double *coefficients, value;

if (polynomial_selected == 1)
    coefficients = (double[3]) {1.5, -3.0, 6.0};
else
    coefficients = (double[3]) {4.5, 1.0, -3.5};
value = evaluate_polynomial(coefficients);
```

Apparentemente questo frammento di programma si comporta nel modo desiderato. La variabile coefficients punta a uno dei due oggetti creati attraverso un letterale composto, e questo oggetto esiste ancora quando la funzione evaluate_polynomial viene invocata. Ora considerate cosa succederebbe se mettessimo delle parentesi graffe attorno alle istruzioni "interne" (quelle controllate dalle istruzioni if):

```
/* Esempio 2 - istruzione if con parentesi graffe */

double *coefficients, value;

if (polynomial_selected == 1) {
    coefficients = (double[3]) {1.5, -3.0, 6.0};
} else {
    coefficients = (double[3]) {4.5, 1.0, -3.5};
}
value = evaluate_polynomial(coefficients);
```

Ora siamo nei guai. Ogni letterale composto crea un oggetto che esiste solamente all'interno del blocco formato dalle parentesi graffe che racchiudono le istruzioni dove compaiono i letterali composti. Nel momento in cui la funzione evaluate_polynomial viene chiamata, la variabile coefficients punta a un oggetto che non esiste più. Il risultato è un comportamento indefinito.

I creatori del C99 non erano contenti di questa situazione perché i programmatore non si aspettavano che la semplice aggiunta di parentesi in un'istruzione if potesse causare un comportamento indefinito. Per evitare questo problema, è stato deciso che le istruzioni interne debbano essere sempre considerate come dei blocchi. Ne risulta che l'esempio 1 e l'esempio 2 sono equivalenti, ovvero entrambi presentano un comportamento indefinito.

Un problema simile può sorgere quando un letterale composto è parte di un'espressione di controllo in un'istruzione di selezione o di un'istruzione di iterazione. Per questa ragione ogni intera istruzione di selezione e di iterazione viene anch'essa considerata un blocco (come se avesse un set di parentesi invisibile attorno all'intera istruzione). Quindi un'istruzione if con una clausola else consiste di tre blocchi: le due istruzioni interne sono considerate dei blocchi così come lo è l'intera istruzione if.

C99

D: La memoria di una variabile con durata di memorizzazione automatica viene allocata quando il blocco che la circonda viene eseguito. Questo è vero anche per i vettori a lunghezza variabile del C99? [p. 476]

R: No. La memoria per i vettori a lunghezza variabile non viene allocata all'inizio del blocco che li circonda perché la lunghezza del vettore non è ancora conosciuta. Viene allocata invece quando l'esecuzione del blocco raggiunge la dichiarazione del vettore. Sotto questo aspetto i vettori a lunghezza variabile sono diversi da tutte le variabili automatiche.

D: Qual è la differenza tra "scope" e "collegamento"? [p.476]

R: Lo scope riguarda il compilatore mentre il collegamento riguarda il linker. Il compilatore usa lo scope di un identificatore per determinare se, in un dato punto del file, sia legale o meno riferirsi all'identificatore stesso. Quando il compilatore traduce un file sorgente in codice oggetto, si annota quali nomi hanno collegamento esterno inserendo eventualmente i loro nomi all'interno di una tabella nel file oggetto. Di conseguenza il linker ha accesso solo ai nomi con collegamento esterno, quelli con collegamento interno e quelli senza collegamento gli sono invisibili.

D: Non capiamo come sia possibile per un nome avere uno scope di blocco e al contempo collegamento esterno. [p. 479]

R: Supponete che un file sorgente definisca una variabile i:

```
int i;
```

Assumete che la definizione di i risieda al di fuori da tutte le funzioni, ne consegue che i ha collegamento esterno per default. In un altro file è presente una funzione f che necessita di accedere a i, così il corpo di f dichiara i come extern:

```
void f(void)
{
    extern int i;
}
```

Nel primo file i ha scope di file. All'interno di f, tuttavia, i ha scope di blocco. Se altre funzioni oltre a f avessero la necessità di accedere a i, dovrebbero dichiararla separatamente (oppure potremmo semplicemente spostare la dichiarazione di i al di fuori di f in modo che abbia scope di file). La confusione è generata dal fatto che ogni dichiarazione o definizione di i stabilisce uno scope diverso: a volte è scope di file, delle altre è scope di blocco.

***D: Perché gli oggetti const non possono essere usati nelle espressioni costanti? const significa costante? [p. 480]**

R: In C const significa "di sola lettura" e non "costante". Guardiamo alcuni esempi che illustrano perché gli oggetti const non possono essere usati nelle espressioni costanti.

Per cominciare un oggetto const può essere costante solo durante la sua esistenza, non durante tutta l'esecuzione del programma. Supponete che un oggetto const venga dichiarato all'interno di una funzione:

```
void f(int n)
{
    const int m = n / 2;
}
```

Quando *f* viene invocata, *m* viene inizializzata al valore di *n* / 2. Il valore di *m* rimarrà costante fino al termine di *f*. Quando *f* viene chiamata la volta dopo, probabilmente a *m* verrà assegnato un valore diverso. Qui sorgono i problemi. Supponete che *m* compaia in un'istruzione *switch*:

```
void f(int n)
{
    const int m = n / 2;

    switch (...) {
        case m: ... /* SBAGLIATO */
    }
}
```

Il valore di *m* non è conosciuto fino al momento in cui *f* viene invocata, il che viola la regola del C che stabilisce che i valori delle etichette debbano essere delle espressioni costanti. Come nuovo esempio guardiamo a un oggetto *const* dichiarato al di fuori di un blocco. Questi oggetti hanno collegamento esterno e possono essere condivisi tra i file. Se il C permettesse l'uso degli oggetti *const* nelle espressioni costanti, ci troveremmo facilmente a dover affrontare la seguente situazione:

```
extern const int n;
int a[n]; /* SBAGLIATO */
```

probabilmente *n* è definita in un altro file rendendo impossibile al compilatore determinare la lunghezza di *a* (stiamo assumendo che *a* sia una variabile esterna e quindi che non possa essere un vettore a lunghezza variabile).

Se questo non è sufficiente a convincervi, considerate quest'altra situazione: se un oggetto *const* viene dichiarato anche *volatile* [qualificatori di tipo volatile > 20.3], il suo valore potrebbe cambiare in ogni momento durante l'esecuzione del programma. Ecco un esempio proveniente dallo standard C:

```
extern const volatile int real_time_clock;
```

La variabile *real_time_clock* non può essere modificata dal programma (perché è stata dichiarata *const*), sebbene il suo valore possa essere modificato attraverso altri meccanismi (perché è dichiarata *volatile*).

D: Perché la sintassi dei dichiaratori è così particolare?

R: Perché è pensata per imitare il suo utilizzo. Il dichiaratore di un puntatore ha la forma **p*, che combacia con il modo nel quale l'operatore asterisco verrà poi applicato a *p*. Il dichiaratore di un vettore ha la forma *a[...]* che combacia con il modo nel quale il

vettore verrà indicizzato. Il dichiaratore di una funzione ha la forma `f(...)` che combacia con la sintassi di una chiamata a una funzione. Questo ragionamento si estende anche ai dichiaratori più complicati. Considerate il vettore `file_cmd` della Sezione 17.7, i cui elementi sono puntatori a funzioni. Il dichiaratore per `file_cmd` ha la forma

`(*file_cmd[])(void)`

e una chiamata a una delle funzioni segue la forma

`(*file_cmd[n])();`

Le parentesi tonde, quelle quadre e il simbolo * si trovano nella stessa posizione.

Esercizi

Sezione 18.1

- Per ognuna delle dichiarazioni seguenti identificate la classe di memorizzazione, i qualificatori di tipo, gli specificatori di tipo, i dichiaratori e gli inizializzatori.
 - `static char **lookup(int level);`
 - `volatile unsigned long io_flags;`
 - `extern char *file_name[MAX_FILES], path[];`
 - `static const char token_buf[] = "";`

Sezione 18.2



- Rispondete a ognuna delle seguenti domande con `auto`, `extern`, `register` e `static`.
 - Quale classe di memorizzazione viene utilizzata principalmente per indicare che una variabile o una funzione può essere condivisa tra molti file?
 - Supponete che la variabile `x` sia condivisa tra diverse funzioni di un file ma nascosta alle funzioni presenti in altri file. Di quale classe di memorizzazione dovrebbe essere dichiarata `x`?
 - Quali classi di memorizzazione possono modificare la durata di memorizzazione di una variabile?
- Elencate la durata di memorizzazione (statica o automatica), lo scope (blocco o file) e il collegamento (interno, esterno o nessuno) di ognuna delle variabili e dei parametri presenti nel seguente file:

```
extern float a;

void f(register double b)
{
    static int c;
    auto char d;
}
```

- Sia `f` la seguente funzione. Quale sarà il valore di `f(10)` se `f` non è mai stata chiamata prima? Quale sarà il valore di `f(10)` nel caso in cui `f` sia stata chiamata cinque volte precedentemente?

```
int f(int i)
{
    static int j = 0;
    return i * j++;
}
```

5. Specificate se ognuna delle seguenti dichiarazioni è vera o falsa. Giustificate le risposte.
- Ogni variabile con durata di memorizzazione statica ha scope di file.
 - Ogni variabile dichiarata all'interno di una funzione non ha collegamento.
 - Ogni variabile con collegamento interno ha durata di memorizzazione statica.
 - Ogni parametro ha scope di blocco.
6. La funzione seguente è pensata per stampare un messaggi di errore. Ogni messaggio viene preceduto da un intero indicante il numero di volte che la funzione è stata chiamata. Sfortunatamente la funzione visualizza sempre 1 come numero del messaggio. Trovate l'errore e spiegate come sistemarlo senza inserire modifiche al di fuori della funzione.

```
void print_error(const char *message)
{
    int n = 1;
    printf("Error %d: %s\n", n++, message);
}
```

- Sezione 18.3** 7. Supponete di dichiarare *x* come un oggetto const. Quale delle seguenti proposizioni su *x* è falsa?
- Se *x* è di tipo int, può essere usata come valore di un'etichetta in un costrutto switch.
 - Il compilatore controllerà che a *x* non venga effettuato alcun assegnamento.
 - x* è soggetta alle stesse regole di scope delle variabili.
 - x* può essere di qualsiasi tipo.

- Sezione 18.4** 8. Scrivete una descrizione completa del tipo di *x* specificato da ognuna delle dichiarazioni seguenti.
- char (*x[10])(int);
 - int (*x(int))[5];
 - float *(*x(void))(int);
 - void (*x(int, void (*y)(int)))(int);
9. Usate una serie di definizioni di tipo per semplificare ognuna delle dichiarazioni dell'Esercizio 8.
10. Scrivete una dichiarazione per le variabili e le funzioni seguenti:
- p* è un puntatore a una funzione con un argomento costituito da un puntatore a carattere che restituisce un puntatore a carattere.
 - f* è una funzione con due argomenti: *p*, un puntatore a una struttura con tag *t*, ed *n*, un intero long. La funzione *f* restituisce un puntatore a una funzione che non ha argomenti e non restituisce nulla.

(c) a è un vettore di quattro puntatori a funzioni che non hanno argomenti e non restituiscono nulla. Gli elementi di a inizialmente puntano a delle funzioni chiamate insert, search, update e print.

(d) b è un vettore di 10 puntatori a funzioni aventi due argomenti di tipo int e che restituiscono strutture con tag t.

11. Nella Sezione 18.4 abbiamo visto che le seguenti dichiarazioni non sono ammissibili:

```
int f(int[]); /* le funzioni non possono restituire vettori */
int g(int)(int); /* le funzioni non possono restituire funzioni */
int a[10](int); /* gli elementi di un vettore non possono essere funzioni */
```

Tuttavia possiamo raggiungere degli effetti simili utilizzando i puntatori: una funzione può restituire un puntatore al primo elemento di un vettore, una funzione può restituire un puntatore a una funzione e gli elementi di un vettore possono essere dei puntatori a funzione. Modificate tutte le dichiarazioni concordemente a quanto detto.

12. *(a) Scrivete una descrizione completa del tipo della funzione f, assumendo che questa sia dichiarata come segue:

```
int (*f(float (*)(long), char *))(double);
```

(b) Fornite un esempio che mostri come verrebbe invocata f.

Sezione 18.5



13. Quali tra le seguenti dichiarazioni solo ammissibili? (Assumete che PI sia una macro che rappresenta il valore 3.14159).

- (a) char c = 65;
- (b) static int i = 5, j = i * i;
- (c) double d = 2 * PI;
- (d) double angles[] = {0, PI / 2, PI, 3 * PI / 2};

14. Quali tipi di variabili non possono essere inizializzate?

- (a) Variabili vettore
- (b) Variabili enumerazione
- (c) Variabili struttura
- (d) Variabili unione
- (e) Nessuna delle precedenti

15. Quale proprietà di una variabile determina se questa abbia o meno un valore iniziale di default?

- (a) Durata di memorizzazione
- (b) Scope
- (c) Collegamento
- (d) Tipo

19 Progettazione di un programma

D&R

È ovvio che i programmi del mondo reale siano più grandi degli esempi presentati in questo libro, tuttavia non potete immaginare quanto grandi siano veramente. CPU più veloci e memorie più capaci hanno reso possibile la scrittura di programmi che sarebbero stati impossibili fino a pochi anni fa. La popolarità delle interfacce grafiche ha incrementato parecchio la lunghezza media dei programmi. La maggior parte dei programmi completi di oggi comprendono almeno 100.000 righe di codice. Programmi costituiti da milioni di righe sono piuttosto comuni e quelli con 10 milioni di righe di codice o più non sono una cosa mai sentita.

Sebbene il C non sia stato pensato per la scrittura di grandi programmi, nella pratica molti di questi sono scritti in C: è un'operazione complicata che richiede una notevole dose di attenzione, tuttavia è fattibile. In questo capitolo discuteremo delle tecniche che si sono dimostrate di aiuto nella scrittura di questo tipo di programmi e vedremo quali funzionalità del C (classe di memorizzazione static, per esempio) risultino particolarmente utili.

La scrittura di programmi di grandi dimensioni (definita spesso come "programmazione in grande") è abbastanza diversa da quella per i piccoli programmi. Equivale alla differenza tra la scrittura di una tesi (10 pagine con interlinea doppia naturalmente) e la scrittura di un libro di 1000 pagine. Un programma di grandi dimensioni richiede più attenzione allo stile, dato che vi lavoreranno molte persone, un'attenta documentazione e la pianificazione della manutenzione, dal momento che probabilmente dovrà essere modificato molte volte.

Dopo tutto, come ha detto Alan Kay (inventore del linguaggio di programmazione Smalltalk) "potete costruire una cuccia per il cane a partire da qualsiasi cosa". Una cuccia può essere costruita senza una particolare progettazione, usando i materiali che si hanno a disposizione. Una casa invece è troppo complessa per essere semplicemente "messa assieme".

Il Capitolo 15 ha trattato la scrittura in C di programmi di grandi dimensioni ma si è concentrato sui dettagli del linguaggio. In questo capitolo riprenderemo l'argomento, concentrando sulle tecniche della buona progettazione del software. Una trattazione completa delle questioni riguardanti la progettazione dei programmi ovviamente esula dagli scopi di questo libro. Tuttavia cercheremo di trattare (brevemen-

te) alcuni concetti importanti nella progettazione di un programma e vedremo come utilizzarli per creare dei programmi C che siano leggibili e manutenibili.

La Sezione 19.1 spiega come vedere un programma C come una collezione di moduli che forniscono l'un l'altro dei servizi. Successivamente vedremo come il concetto di information hiding (Sezione 19.2) e tipi di dato astratti (Sezione 19.3) possono migliorare questi moduli. Concentrandoci su un singolo esempio (un tipo di dato stack), la Sezione 19.4 illustra come un tipo di dato astratto può essere definito e implementato in C. La Sezione 19.5 descrive alcune limitazioni del C nel definire dei tipi di dati astratti e mostra come aggirarle.

19.1 Moduli

Spesso, quando si progetta un programma in C (o in un altro linguaggio di programmazione), è utile vederlo come costituito da un certo numero di **moduli** indipendenti. Un modulo è costituito da una collezione di servizi, alcuni dei quali devono essere resi disponibili alle altre parti del programma (i cosiddetti **client**). Ogni modulo possiede un'**interfaccia** che descrive i servizi disponibili. I dettagli del modulo (incluso il codice sorgente per gli stessi servizi) sono contenuti nell'**implementazione** del modulo stesso.

Nel contesto del C, i "servizi" sono le funzioni. L'interfaccia di un modulo è il file header che contiene i prototipi delle funzioni che sono rese disponibili ai client (i file sorgente). L'implementazione di un modulo è il file sorgente che contiene le definizioni delle funzioni del modulo stesso.

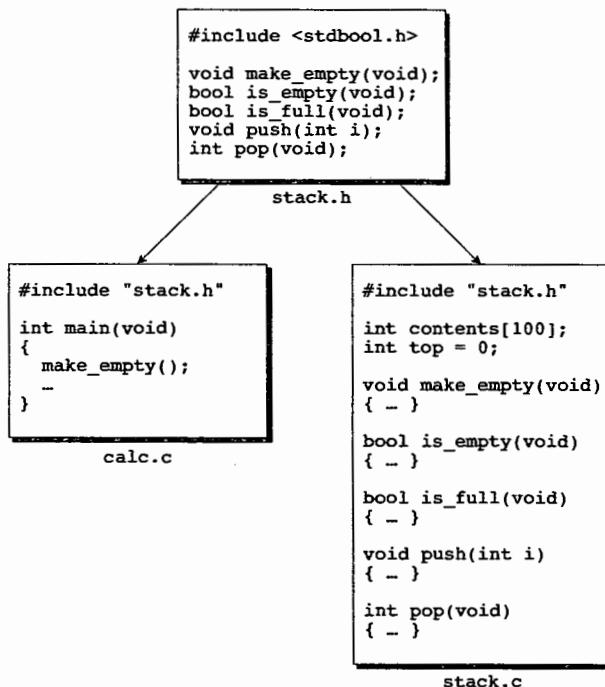
Per illustrare questa terminologia, osserviamo il programma calcolatrice che è stato abbozzato nella Sezione 15.1 e nella Sezione 15.2. Questo programma è composto dal file calc.c che contiene la funzione main, e dal modulo stack, che è contenuto nei file stack.h e stack.c (si veda la figura in cima alla prossima pagina). Il file calc.c è un client del modulo stack. Il file stack.h è invece l'interfaccia del modulo che fornisce ai client tutto ciò di cui hanno bisogno di sapere circa il modulo. Il file stack.c è l'implementazione del modulo che contiene le definizioni delle funzioni dello stack assieme alle dichiarazioni delle variabili che lo costituiscono.

La libreria del C è a sua volta una collezione di moduli. Ogni header presente della libreria funge da interfaccia per un modulo. L'header <stdio.h> per esempio, è l'interfaccia a un modulo contenente le funzioni I/O, mentre <string.h> è l'interfaccia per un modulo contenente le funzioni per la manipolazione delle stringhe.

Suddividere un programma in moduli presenta diversi vantaggi.

- **Astrazione.** Se i moduli sono progettati adeguatamente, possiamo trattarli come delle **astrazioni**. Sappiamo quello che fanno, ma non ci preoccupiamo dei dettagli riguardanti il come lo fanno. Grazie all'astrazione, per modificare una parte, non è necessario capire come funziona l'intero programma. L'astrazione, inoltre, rende più facile lavorare sullo stesso programma da parte di diversi membri di un gruppo. Una volta trovato l'accordo sulle interfacce dei moduli, la responsabilità di implementare ogni modulo può essere delegata a una particolare persona. I membri di un gruppo possono lavorare indipendentemente gli uni dagli altri.

- **Riusabilità.** Qualsiasi modulo che fornisca dei servizi è potenzialmente riutilizzabile in altri programmi. Il nostro modulo stack, per esempio, può essere riutilizzato. Spesso è difficile prevedere gli usi futuri di un modulo, per questo è buona pratica progettarli nell'ottica della riusabilità.
- **Manutenibilità.** Solitamente un piccolo baco ha effetto solo su un singolo modulo dell'implementazione e questo rende il baco più facile da localizzare e correggere. Una volta che il baco è stato corretto, rifare il build del programma richiede solamente la ricompilazione del modulo interessato (seguita dal linking dell'intero programma). Su larga scala possiamo anche sostituire l'implementazione di un intero modulo, per esempio per migliorare le performance o per fare il porting su un'altra piattaforma.



Sebbene tutti questi vantaggi siano importanti, la manutenibilità è di gran lunga il vantaggio più importante. La maggior parte dei programmi del mondo reale rimangono in servizio per anni, durante i quali vengono scoperti bachi, vengono apportati miglioramenti e vengono eseguite modifiche per andare incontro al mutare delle specifiche. Progettare un programma in modo modulare rende la manutenzione molto più facile. La sostituzione di una ruota bucata non dovrebbe comportare la revisione del motore.

Per fare un esempio non abbiamo bisogno di andare più lontano del programma `inventory` dei Capitoli 16 e 17. Il programma originale (Sezione 16.3) salvava i componenti in un vettore. Supponete che il cliente, dopo aver utilizzato il programma per un certo periodo, si lamenti del fatto che il programma presenti un limite sul numero di componenti salvabili. Per soddisfare le richieste del cliente, potremmo passare a una lista concatenata (come abbiamo fatto nella Sezione 17.5). Effettuare questa modifica richiede di cercare all'interno del programma tutti i punti nei quali c'è una dipendenza nel modo in cui i componenti vengono salvati. Se avessimo progettato il programma in modo diverso fin da principio, ovvero con un modulo separato che gestisce il salvataggio dei componenti, avremmo avuto bisogno di riscrivere solamente l'implementazione di quel modulo e non l'intero programma.

Una volta convinti che la progettazione modulare sia la strada giusta, la progettazione del programma procede decidendo quali moduli lo debbano costituire, quali servizi questi debbano presentare e come i moduli debbano essere collegati. Ora tratteremo brevemente questi argomenti.

Coesione e accoppiamento

Delle buone interfacce per i moduli non sono costituite da collezioni casuali di dichiarazioni. In un programma ben progettato, i moduli devono possedere due proprietà.

- **Alta coesione.** Gli elementi di ogni modulo devono essere strettamente collegati gli uni agli altri. Possiamo pensarli come se cooperassero per raggiungere un obiettivo comune. Una grande coesione rende i moduli più semplici da usare e rende l'intero programma più semplice da capire.
- **Basso accoppiamento.** I moduli dovrebbero essere il più possibile indipendenti tra loro. Un basso accoppiamento facilita la modifica di un programma e il riutilizzo dei moduli.

Il programma calcolatrice presenta queste proprietà? Il modulo `stack` è chiaramente coeso: le sue funzioni rappresentano le operazioni di uno stack. Nel programma c'è poco accoppiamento. Il file `calc.c` dipende da `stack.h` (e `stack.c` dipende da `stack.h`, ovviamente) ma non ci sono altre dipendenze appariscenti.

Tipi di moduli

A causa della necessità di avere un'alta coesione e un basso accoppiamento, i moduli tendono a ricadere all'interno di alcune categorie tipo.

- Un **data pool** è una collezione di variabili e/o costanti tra loro affini. In C, un modulo di questo tipo è spesso costituito da un solo file header. Solitamente, dal punto di vista della programmazione, mettere le variabili nei file header non è una buona idea. Tuttavia mettere delle costanti collegate tra loro in un file header si rivela spesso utile. Nella libreria del C, `<float.h>` [header `<float.h>` > 23.1] e `<limits.h>` [header `<limits.h>` > 23.2] sono degli esempi di data pool.
- Una **libreria** è una collezione di funzioni affini. L'header `<string.h>`, per esempio, è l'interfaccia per le funzioni di libreria per la gestione delle stringhe.

- Un **oggetto astratto** è costituito da una collezione di funzioni che operano su una struttura dati nascosta. In questo capitolo il termine oggetto assume un significato diverso rispetto al resto del libro. Nella terminologia del C, un oggetto è semplicemente un blocco di memoria che può contenere un valore. In questo capitolo, però, un oggetto è una collezione di dati raggruppata assieme a delle operazioni sui dati stessi. Se i dati sono nascosti, l'oggetto è "astratto". Il modulo stack di cui stiamo discutendo appartiene a questa categoria.
- Un **tipo di dato astratto** (detto anche abstract data type o ADT) è un tipo la cui rappresentazione è nascosta. I moduli client possono usare il tipo per dichiarare delle variabili, ma non conoscono in alcun modo la struttura di queste. Per eseguire un'operazione su variabili di questo tipo, un modulo client deve chiamare una funzione fornita dal modulo del tipo di dato astratto. I tipi di dato astratti giocano un ruolo significativo nella moderna programmazione, ritorneremo su questo argomento nelle sezioni dalla 19.3 alla 19.5.

19.2 Information hiding

Spesso un modulo ben progettato mantiene alcune informazioni segrete nei confronti dei suoi client. I client del nostro modulo stack, per esempio, non hanno alcuna necessità di sapere se lo stack sia contenuto in un vettore o in una lista concatenata o in qualche altra forma ancora. Nascondere deliberatamente alcune informazioni ai client di un modulo è conosciuto come **information hiding**. L'information hiding presenta principalmente due vantaggi.

- **Sicurezza**. Se i client non conoscono come viene memorizzato lo stack, non saranno in grado di corromperlo manomettendo il suo funzionamento interno. Per eseguire delle operazioni sullo stack, sono costretti a chiamare le funzioni che vengono fornite dallo stesso modulo (funzioni che abbiamo scritto e testato).
- **Flessibilità**. Non sarà difficile effettuare delle modifiche (non importa quando grandi) al funzionamento interno di un modulo. Per esempio, inizialmente possiamo implementare lo stack come un vettore e poi successivamente passare a una lista concatenata o a qualche altro tipo di rappresentazione. Naturalmente dovremo riscrivere l'implementazione del modulo ma, se il modulo è stato concepito correttamente, non dovremo modificare la sua interfaccia.

In C, lo strumento principale per forzare l'information hiding è la classe di memorizzazione static [classe di memorizzazione statica > 18.2]. Dichiarare una variabile con scope di file come static le assegna un collegamento interno, il che la ripara dall'essere accessibile da altri file, inclusi i client del modulo (dichiarare una funzione come static è anche utile, la funzione può essere chiamata direttamente solo dalle funzioni presenti nello stesso file).

Un modulo stack

Per vedere i benefici dell'information hiding, guardiamo a due implementazioni di un modulo stack: la prima usando un vettore, la seconda usando una lista concatenata. Il file header del modulo si presenta in questo modo:

```
stack.h #ifndef STACK_H
#define STACK_H

#include <stdbool.h> /* solo C99 */

void make_empty(void);
bool is_empty(void);
bool is_full(void);
void push(int i);
int pop(void);

#endif
```

Abbiamo incluso l'header <stdbool.h> del C99 in modo che le funzioni `is_empty` e `is_full` possano restituire un risultato `bool` invece che un valore `int`.

Usiamo inizialmente l'implementazione dello stack:

```
stack1.c #include <stdio.h>
#include <stdlib.h>
#include "stack.h"

#define STACK_SIZE 100

static int contents[STACK_SIZE];
static int top = 0;

static void terminate(const char *message)
{
    printf("%s\n", message);
    exit(EXIT_FAILURE);
}

void make_empty(void)
{
    top = 0;
}

bool is_empty(void)
{
    return top == 0;
}

bool is_full(void)
{
    return top == STACK_SIZE;
}

void push(int i)
{
    if (is_full())
        terminate("Error in push: stack is full.");
    contents[top++] = i;
}
```

```
int pop(void)
{
    if (is_empty())
        terminate("Error in pop: stack is empty.");
    return contents[--top];
}
```

Le variabili che costituiscono lo stack (contents e top) sono entrambe dichiarate static dato che non c'è nessuna ragione per la quale il resto di un programma debba accedervi direttamente. Anche la funzione terminate viene dichiarata static. Questa funzione non fa parte dell'interfaccia del modulo, invece è stata progettata per essere usata solamente all'interno dell'implementazione di un modulo.

Per una questione di stile alcuni programmatore utilizzano delle macro per indicare quali funzioni e quali variabili sono "pubbliche" (accessibili altrove nel programma) e quali sono "private" (limitate a un singolo file):

```
#define PUBLIC /* vuoto */
#define PRIVATE static
```

La ragione per scrivere PRIVATE invece di static è che quest'ultimo ha più di uno scopo nel C. PRIVATE rende chiaro che lo stiamo usando per imporre l'information hiding. Ecco come si presenterebbe l'implementazione dello stack nel caso in cui utilizzassimo le macro PUBLIC e PRIVATE:

```
PRIVATE int contents[STACK_SIZE];
PRIVATE int top = 0;

PRIVATE void terminate(const char *message) { ... }

PUBLIC void make_empty(void) { ... }

PUBLIC bool is_empty(void) { ... }

PUBLIC bool is_full(void) { ... }

PUBLIC void push(int i) { ... }

PUBLIC int pop(void) { ... }
```

Ora passeremo all'implementazione del modulo stack basata su una lista concatenata:

```
stack2.c #include <stdio.h>
#include <stdlib.h>
#include "stack.h"

struct node {
    int data;
    struct node *next;
};
```

```

static struct node *top = NULL;
static void terminate(const char *message)
{
    printf("%s\n", message);
    exit(EXIT_FAILURE);
}

void make_empty(void)
{
    while (!is_empty())
        pop();
}

bool is_empty(void)
{
    return top == NULL;
}

bool is_full(void)
{
    return false;
}

void push(int i)
{
    struct node *new_node = malloc(sizeof(struct node));
    if (new_node == NULL)
        terminate("Error in push: stack is full.");
    new_node->data = i;
    new_node->next = top;
    top = new_node;
}

int pop(void)
{
    struct node *old_top;
    int i;

    if (is_empty())
        terminate("Error in pop: stack is empty.");

    old_top = top;
    i = top->data;
    top = top->next;
    free(old_top);
    return i;
}

```

Osservate che la funzione `is_full` restituisce il valore `false` ogni volta che viene chiamata. Una lista concatenata non ha limiti alle sue dimensioni, di conseguenza lo stack non sarà mai pieno. È possibile (ma non probabile) che il programma possa esaurire

la memoria, il che causerebbe il fallimento della funzione `push`, ma non c'è un modo facile per controllare in anticipo questa eventualità.

Il nostro esempio dello stack, illustra chiaramente i vantaggi dell'information hiding: non ha importanza se utilizziamo `stack1.c` o `stack2.c` per implementare il modulo di stack. Entrambe le versioni combaciano con l'interfaccia del modulo e quindi possiamo passare da uno all'altro senza dover effettuare modifiche in altri punti del programma.

19.3 Tipi di dato astratti

Un modulo che, come lo stack della sezione precedente, funga da oggetto astratto possiede uno svantaggio serio: non esiste un modo per avere istanze multiple dell'oggetto (più di uno stack in questo caso). Per ottenere questo abbiamo bisogno di fare un passo avanti e creare un nuovo *tipo*.

Una volta che abbiamo definito il tipo `Stack`, siamo in grado di avere tutti gli stack di cui abbiamo voglia. Il seguente frammento illustra come possiamo avere due stack nello stesso programma:

```
Stack s1, s2;  
  
make_empty(&s1);  
make_empty(&s2);  
push(&s1, 1);  
push(&s2, 2);  
if (!is_empty(&s1))  
    printf("%d\n", pop(&s1)); /* stampa "1" */
```

Non sappiamo cosa siano effettivamente `s1` ed `s2` (strutture? puntatori?) ma questo non ha alcuna importanza. Per i client, `s1` ed `s2` sono delle astrazioni che rispondono a certe operazioni (`make_empty`, `is_empty`, `is_full`, `push` e `pop`).

Convertiamo il nostro header `stack.h` in modo che fornisca un tipo `Stack`, dove quest'ultimo è una struttura. Fare ciò richiede l'aggiunta di un parametro `Stack` (o `Stack*`) a ogni funzione. Ora l'header si presenterà in questo modo (le modifiche a `stack.h` sono in **grassetto**, le parti non modificate dell'header non vengono mostrate):

```
#define STACK_SIZE 100  
  
typedef struct {  
    int contents[STACK_SIZE];  
    int top;  
} Stack;  
  
void make_empty(Stack *s);  
bool is_empty(const Stack *s);  
bool is_full(const Stack *s);  
void push(Stack *s, int i);  
int pop(Stack *s);
```

I parametri Stack alle funzioni `make_empty`, `push` e `pop` devono essere dei puntatori dato che queste funzioni modificano lo stack. I parametri `is_empty` e `is_full` non necessitano di essere dei puntatori, ma sono stati resi tali comunque. Passare a queste funzioni un puntatore a Stack invece che un valore Stack è più efficiente dato che quest'ultimo comporterebbe la copia di una struttura.

Incapsulamento

Sfortunatamente Stack non è un tipo di dato astratto visto che `stack.h` rivela cosa sia effettivamente il tipo Stack. Nulla previene i client dall'usare una variabile Stack come una struttura:

```
Stack s1;
s1.top = 0;
s1.contents[top++] = 1;
```

Fornire un accesso ai membri `top` e `contents` permette ai client di corrompere lo stack. Peggio ancora, non saremo in grado di modificare il modo in cui gli stack vengono memorizzati senza doverci assicurare delle ripercussioni che la modifica ha sui client.

Quello di cui abbiamo bisogno è un modo per evitare che i client conoscano com'è rappresentato il tipo Stack. Il C possiede solamente un supporto limitato per **incapsulare** i tipi in questo modo. I linguaggi più recenti basati sul C, tra cui il C++, Java e C# sono meglio equipaggiati a questo scopo.

Tipi incompleti

L'unico strumento che il C ci fornisce per l'incapsulamento dei dati è costituito dai **tipi incompleti** (i tipi incompleti sono stati menzionati brevemente nella Sezione 17.9 e nella Sezione Domande & Risposte alla fine del Capitolo 17). Lo standard C descrive i tipi incompleti come "i tipi che descrivono oggetti ma che mancano delle informazioni necessarie a determinare la loro dimensione". Per esempio, la dichiarazione



```
struct t; /* dichiarazione incompleta di t */
```

dice al compilatore che `t` è un tag di struttura ma non descrive i membri di quest'ultima. Ne risulta che il compilatore non possiede informazioni sufficienti per determinare la dimensione di una struttura di questo tipo. L'intento è che il tipo incompleto venga completato altrove all'interno del programma.

Fintanto che il tipo rimane incompleto, i suoi usi sono limitati. Dal momento che il compilatore non conosce la dimensione di un tipo incompleto, questo non può essere usato per dichiarare una variabile:

```
struct t s; /** SBAGLIATO **/
```

Tuttavia è perfettamente ammissibile definire un tipo puntatore che si riferisce a un tipo incompleto:

```
typedef struct t *T;
```

Questo tipo di definizione stabilisce che la variabile del tipo `T` è un puntatore a una struttura con tag `t`. Adesso possiamo dichiarare delle variabili di tipo `T`, passarle come argomenti alle funzioni ed eseguire altre operazioni che siano ammissibili per i puntatori (la dimensione di un puntatore non dipende da quello a cui punta, il che spiega perché il C permette questo tipo di comportamento). Quello che non possiamo fare è applicare l'operatore `>` a una di queste variabili, dato che il compilatore non sa nulla dei membri di una struttura `t`.

19.4 Un tipo di dato astratto per lo stack

Per illustrare come i tipi di dato astratti possano essere incapsulati usando i tipi incompleti, svilupperemo uno stack ADT basato sul modulo descritto nella Sezione 19.2. Nel farlo esploreremo tre modi diversi per implementare lo stack.

Definire l'interfaccia per lo stack ADT

Per prima cosa abbiamo bisogno di un file header che definisca il nostro tipo stack ADT e fornisca i prototipi per le funzioni che rappresentano le operazioni sullo stack. Chiamiamo questo file `stackADT.h`. Il tipo `Stack` sarà un puntatore a una struttura `stack_type` che manterrà i contenuti attuali dello stack. Questa struttura è un tipo incompleto che verrà completato nel file che implementa lo stack. I membri di queste strutture dipenderanno da come lo stack è implementato. Ecco come si presenterà il file `stackADT.h`:

```
stackADT.h
(versione 1)
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h> /* solo C99 */

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, int i);
int pop(Stack s);

#endif
```

I client che includeranno il file `stackADT.h` saranno in grado di dichiarare delle variabili di tipo `Stack`, ognuna delle quali sarà in grado di puntare a una struttura `stack_type`. I client potranno così chiamare le funzioni dichiarate in `stackADT.h` per eseguire le operazioni sulle variabili `stack`. Tuttavia i client non possono accedere ai membri della struttura `stack_type` visto che quella struttura verrà definita in un file separato.

Osservate che ogni funzione ha un parametro `Stack` o restituisce un valore `Stack`. Le funzioni dello stack della Sezione 19.3 possedevano parametri di tipo `Stack *`. La ragione per questa differenza è che adesso la variabile `Stack` è un puntatore, punta a una struttura `stack_type` che mantiene i contenuti dello stack. Se una funzione ha

bisogno di modificare lo stack, questa modifica la struttura stessa, non il puntatore alla struttura.

Osservate anche la presenza delle funzioni `create` e `destroy`. Un modulo generalmente non ha bisogno di queste funzioni, tuttavia questo accade per un modulo ADT. La funzione `create` allocherà dinamicamente della memoria per lo stack (inclusa la memoria richiesta per una struttura `stack_type`), così come inizializzerà lo stack nel suo stato "vuoto". La funzione `destroy` rilascerà la memoria dello stack che era stata allocata dinamicamente.

Il seguente file client può essere usato per testare lo stack ADT. Crea due stack ed esegue una serie di operazioni su di essi.

```
stackclient.c #include <stdio.h>
#include "stackADT.h"

int main(void)
{
    Stack s1, s2;
    int n;

    s1 = create();
    s2 = create();

    push(s1, 1);
    push(s1, 2);

    n = pop(s1);
    printf("Popped %d from s1\n", n);
    push(s2, n);
    n = pop(s1);
    printf("Popped %d from s1\n", n);
    push(s2, n);

    destroy(s1);

    while (!is_empty(s2))
        printf("Popped %d from s2\n", pop(s2));

    push(s2, 3);
    make_empty(s2);
    if (is_empty(s2))
        printf("s2 is empty\n");
    else
        printf("s2 is not empty\n");

    destroy(s2);

    return 0;
}
```

Se lo stack ADT viene implementato correttamente, il programma dovrebbe produrre il seguente output:

```
Popped 2 from s1
Popped 1 from s1
Popped 1 from s2
Popped 2 from s2
s2 is empty
```

Implementare lo stack ADT usando un vettore di lunghezza fissa

Ci sono diversi modi per implementare lo stack ADT. Il primo approccio che adotteremo è il più semplice. Faremo in modo che il file `stackADT.c` definisca la struttura `stack_type` in modo che contenga un vettore di lunghezza fissa (per conservare i contenuti del vettore) assieme a un intero che tiene traccia della cima dello stack:

```
struct stack_type {
    int contents[STACK_SIZE];
    int top;
};
```

Ecco come si presenterà il file `stackADT.c`:

```
stackADT.c #include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

#define STACK_SIZE 100

struct stack_type {
    int contents[STACK_SIZE];
    int top;
};

static void terminate(const char *message)
{
    printf("%s\n", message);
    exit(EXIT_FAILURE);
}

Stack create(void)
{
    Stack s = malloc(sizeof(struct stack_type));
    if (s == NULL)
        terminate("Error in create: stack could not be created.");
    s->top = 0;
    return s;
}

void destroy(Stack s)
{
    free(s);
}
```

```

void make_empty(Stack s)
{
    s->top = 0;
}

bool is_empty(Stack s)
{
    return s->top == 0;
}

bool is_full(Stack s)
{
    return s->top == STACK_SIZE;
}

void push(Stack s, int i)
{
    if (is_full(s))
        terminate("Error in push: stack is full.");
    s->contents[s->top++] = i;
}

int pop(Stack s)
{
    if (is_empty(s))
        terminate("Error in pop: stack is empty.");
    return s->contents[--s->top];
}

```

La cosa più affascinante a riguardo delle funzioni di questo file è che queste utilizzano l'operatore `->` e non l'operatore `.` per accedere ai membri `contents` e `top` della struttura `stack_type`. Il parametro `s` è un puntatore a una struttura `stack_type` e non una struttura stessa, di conseguenza l'uso dell'operatore `.` non sarebbe ammissibile.

Modificare il tipo degli elementi dello stack ADT

Ora abbiamo una versione funzionante, cerchiamo di migliorarla. Per prima cosa osservate che gli elementi dello stack devono essere interi. Questo è troppo restrittivo, infatti il tipo degli elementi non ha alcuna importanza. Gli elementi contenuti nello stack potrebbero essere di un altro dei tipi base (`float`, `double`, `long`, etc) o anche strutture, unioni o puntatori.

Per rendere lo stack più facile da modificare per i diversi tipi degli elementi, aggiungiamo una definizione di tipo all'header `stackADT.h`. Definiremo un tipo chiamato `Item` che rappresenterà il tipo degli elementi contenuti nello stack.

```

stackADT.h
(versione 2)

#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h> /* C99 only */

typedef int Item;

```

```

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, Item i);
Item pop(Stack s);

#endif

```

Le modifiche apportate al file sono indicate in grassetto. Oltre all'aggiunta del tipo `Item`, sono state modificate le funzioni `push` e `pop`. Ora `push` ha un parametro di tipo `Item`, mentre `pop` restituisce un valore di tipo `Item`. D'ora in avanti utilizzeremo questa versione di `stackADT.h`.

Il file `stackADT.c` deve essere modificato in accordo al nuovo header. Le modifiche, tuttavia, sono minimi. Ora la struttura `stack_type` contiene un vettore i cui elementi sono di tipo `Item` invece che `int`.

```

struct stack_type {
    Item contents[STACK_SIZE];
    int top;
};

```

Le uniche altre modifiche sono sulle funzioni `push` (ora il secondo parametro è di tipo `Item`) e `pop` (che restituisce un valore di tipo `Item`). Il corpo di queste funzioni non viene modificato.

Il file `stackclient.c` può essere usato come test per i nuovi `stackADT.h` e `stackADT.c` in modo da verificare che il tipo `Stack` funzioni ancora (lo fa!). Ora possiamo modificare il tipo degli elementi tutte le volte che vogliamo, modificando semplicemente la definizione del tipo `Item` presente all'interno di `stackADT.h` (anche se non dovremo modificare il file `stackADT.c`, lo dovremo ricompilare comunque).

Implementare lo stack ADT usando un vettore dinamico

Un altro problema con l'attuale implementazione dello stack ADT è dato dal fatto che ogni stack possiede una dimensione massima che correntemente è fissata a 100 elementi. Naturalmente possiamo incrementare il limite fino a raggiungere qualsiasi valore vogliamo, tuttavia tutti gli stack creati usando il tipo `Stack` avranno lo stesso limite. Non c'è modo di avere stack con diversa capacità o imporre la dimensione dello stack mentre il programma è in esecuzione.

Ci sono due soluzioni di questo problema. Una di queste è implementare lo stack come una lista concatenata, nel qual caso non ci sarà una dimensione prefissata per le dimensioni. Tra una attimo investigheremo questa soluzione. Prima però proveremo un altro approccio che coinvolge il salvataggio degli elementi in un vettore allocato dinamicamente [vettori allocati dinamicamente > 17.3].

Il problema di questo secondo approccio è quello di modificare la struttura stack_type in modo che il membro contents sia un puntatore a un vettore nel quale vengono contenuti gli elementi e non il vettore stesso:

```
struct stack_type {
    Item *contents;
    int top;
    int size;
};
```

Abbiamo aggiunto anche un nuovo membro, chiamato size, che contiene la dimensione massima dello stack (la lunghezza del vettore puntato da contents). Utilizzeremo questo membro per controllare la condizione di "stack pieno".

Ora la funzione create avrà un parametro che specifica la dimensione massima desiderata:

```
Stack create(int size);
```

Quando la funzione create viene invocata, crea una struttura stack_type più un vettore di lunghezza size. Il membro contents della struttura punterà a questo vettore.

Il file stackADT.h sarà uguale al precedente, a eccezione del fatto che dovremo aggiungere il parametro size alla funzione create (chiameremo la nuova versione stackADT2.h). Il file stackADT.c avrà bisogno invece di una modifica più estensiva. La nuova versione compare di seguito con le modifiche contrassegnate in grassetto.

```
stackADT2.c
#include <stdio.h>
#include <stdlib.h>
#include "stackADT2.h"

struct stack_type {
    Item *contents;
    int top;
    int size;
};

static void terminate(const char *message)
{
    printf("%s\n", message);
    exit(EXIT_FAILURE);
}

Stack create(int size)
{
    Stack s = malloc(sizeof(struct stack_type));
    if (s == NULL)
        terminate("Error in create: stack could not be created.");
    s->contents = malloc(size * sizeof(Item));
    if (s->contents == NULL) {
        free(s);
        terminate("Error in create: stack could not be created.");
    }
}
```

```
s->top = 0;
s->size = size;
return s;
}

void destroy(Stack s)
{
    free(s->contents);
    free(s);
}

void make_empty(Stack s)
{
    s->top = 0;
}

bool is_empty(Stack s)
{
    return s->top == 0;
}

bool is_full(Stack s)
{
    return s->top == s->size;
}

void push(Stack s, Item i)
{
    if (is_full(s))
        terminate("Error in push: stack is full.");
    s->contents[s->top++] = i;
}

Item pop(Stack s)
{
    if (is_empty(s))
        terminate("Error in pop: stack is empty.");
    return s->contents[--s->top];
}
```

Adesso la funzione `create` chiama la `malloc` due volte: una per allocare una struttura `stack_type` e una per allocare il vettore che conterrà gli elementi dello stack. Entrambe le chiamate alla funzione `malloc` possono fallire causando la chiamata della funzione `terminate`. La funzione `destroy` deve chiamare la funzione `free` due volte per rilasciare tutta la memoria allocata dalla `create`.

Il file `stackclient.c` può essere nuovamente usato per testare lo stack ADT. Tuttavia le chiamate alla `create` dovranno essere modificate dato che ora la funzione `create` richiede un argomento. Per esempio possiamo rimpiazzare le istruzioni

```
s1 = create();
s2 = create();
```

con quelle seguenti:

```
s1 = create(100);
s2 = create(200);
```

Implementare lo stack ADT usando una lista concatenata

Implementare lo stack con un vettore allocato dinamicamente ci fornisce maggiore flessibilità rispetto all'uso di un vettore a lunghezza fissa. Tuttavia il client ha ancora bisogno di specificare la dimensione massima dello stack nel momento in cui questo viene creato. Se usassimo una lista concatenata non ci sarebbe alcun limite predefinito alla dimensione dello stack.

La nostra implementazione sarà simile a quella del file stack2.c della Sezione 19.2. La lista concatenata consisterà di nodi rappresentati dalla seguente struttura:

```
struct node {
    Item data;
    struct node *next;
};
```

Adesso il membro `data` è di tipo `Item` invece che di tipo `int`, ma per il resto la struttura è la stessa.

La struttura `stack_type` conterrà un puntatore al primo nodo della lista:

```
struct stack_type {
    struct node *top;
};
```

A prima vista la struttura `stack_type` sembra superflua: potremmo semplicemente definire `Stack` del tipo `struct node *` e lasciare che il suo valore sia un puntatore al primo nodo della lista. Tuttavia abbiamo ancora bisogno della struttura `stack_type` in modo che l'interfaccia allo stack rimanga la stessa (se la togliessimo ogni funzione che modifica lo stack avrebbe bisogno di un parametro di tipo `Stack *` invece che di un parametro di tipo `Stack`). Inoltre, avere la struttura `stack_type` facilita eventuali modifiche all'implementazione nel caso in cui decidessimo di aggiungere delle informazioni aggiuntive. Per esempio, se in un secondo momento decidessimo che la struttura `stack_type` dovesse contenere un contatore di quanti elementi sono contenuti correntemente nello stack, potremmo facilmente aggiungere un membro per contenere queste informazioni.

Non abbiamo bisogno di effettuare modifiche all'header `stackADT.h` (useremo questo file e non `stackADT2.h`). Per il testing possiamo anche usare il file `stackclient.c` originale. Tutte le modifiche verranno fatte all'interno del file `stackADT.c`. Ecco la nuova versione:

```
stackADT3.c
#include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"
```

```
struct node {
    Item data;
    struct node *next;
};

struct stack_type {
    struct node *top;
};

static void terminate(const char *message)
{
    printf("%s\n", message);
    exit(EXIT_FAILURE);
}

Stack create(void)
{
    Stack s = malloc(sizeof(struct stack_type));
    if (s == NULL)
        terminate("Error in create: stack could not be created.");
    s->top = NULL;
    return s;
}

void destroy(Stack s)
{
    make_empty(s);
    free(s);
}

void make_empty(Stack s)
{
    while (!is_empty(s))
        pop(s);
}

bool is_empty(Stack s)
{
    return s->top == NULL;
}

bool is_full(Stack s)
{
    return false;
}

void push(Stack s, Item i)
{
    struct node *new_node = malloc(sizeof(struct node));
    if (new_node == NULL)
        terminate("Error in push: stack is full.");
```

```

    new_node->data = i;
    new_node->next = s->top;
    s->top = new_node;
}
Item pop(Stack s)
{
    struct node *old_top;
    Item i;

    if (is_empty(s))
        terminate("Error in pop: stack is empty.");

    old_top = s->top;
    i = old_top->data;
    s->top = old_top->next;
    free(old_top);
    return i;
}

```

Osservate come la funzione `destroy` chiama la funzione `make_empty` (per rilasciare la memoria occupata dai nodi nella lista concatenata) prima di chiamare la `free` (per rilasciare la memoria di una struttura `stack_type`).

19.5 Elementi di progettazione per i tipi di dato astratti

La Sezione 19.4 descrive uno stack ADT e presenta diversi modi per implementarlo. Sfortunatamente questa struttura ADT soffre di seri problemi che non la rendono robusta. Guardiamo a ognuno di questi problemi e discutiamo delle possibili soluzioni.

Convenzioni sui nomi

Attualmente le funzioni per lo stack ADT hanno dei nomi corti e facilmente comprensibili: `create`, `destroy`, `make_empty`, `is_empty`, `is_full`, `push` e `pop`. Se nel programma abbiamo più di una struttura ADT, le collisioni tra nomi diventano probabili con le funzioni di due moduli aventi lo stesso nome (ogni ADT avrà bisogno della sua funzione `create`, per esempio). Di conseguenza, probabilmente avremo bisogno di usare dei nomi di funzione che incorporano il nome della stessa ADT, come `stack_create` al posto di `create`.

Gestione degli errori

Lo stack ADT gestisce gli errori visualizzando un messaggio e facendo terminare il programma. Questa non è una cosa sbagliata da fare. Il programmatore può evitare l'estrazione di elementi da uno stack vuoto e l'inserimento di elementi in uno stack pieno chiamando diligentemente la funzione `is_empty` prima di ogni chiamata alla `pop`, e la funzione `is_full` prima di ogni chiamata alla `push`. Quindi in teoria non c'è motivo per cui le funzioni `push` e `pop` debbano fallire (nell'implementazione con la

lista concatenata però la chiamata alla `is_full` non è a prova di stupido: una successiva chiamata alla `push` può fallire comunque). Nonostante ciò potremmo voler fornire al programma un modo per riprendersi da questi errori invece che terminare.

Un'alternativa è di avere delle funzioni `push` e `pop` che restituiscono un valore `bool` che indichi se queste abbiano avuto successo o meno. Attualmente la funzione `push` ha `void` come tipo restituito, di conseguenza possiamo modificarla facilmente per fare in modo che restituisca il valore `true` nel caso in cui l'operazione di inserimento abbia successo e `false` nel caso in cui lo stack sia pieno. Modificare la funzione `pop` è più complesso dato che attualmente questa funzione restituisce il valore che è stato prelevato. Tuttavia se la funzione restituisse, invece del valore prelevato, un puntatore a quest'ultimo, allora nel caso in cui lo stack fosse vuoto potrebbe utilizzare `NULL` come valore restituito.

Un commento finale sulla gestione degli errori: la libreria dello standard C contiene una macro parametrica chiamata `assert` [macro assert > 24.1] che termina il programma nel caso in cui la condizione specificata non venisse soddisfatta. Possiamo utilizzare delle chiamate a questa macro in sostituzione alle istruzioni `if` e alle chiamate alla funzione `terminate` che compaiono attualmente nello stack ADT.

ADT generici

A metà della Sezione 19.4 abbiamo migliorato lo stack ADT rendendo più facile la modifica del tipo degli elementi contenuti. Tutto quello che dovevamo fare era modificare la definizione del tipo `Item`, tuttavia doverlo fare rappresentava comunque una seccatura. Sarebbe stato meglio se lo stack avesse potuto contenere elementi di qualsiasi tipo senza dover modificare il file `stack.h`. Osservate anche che il nostro stack ADT soffre di un serio problema: un programma non può creare due stack i cui elementi sono di tipo diverso. È facile creare diversi stack, ma questi devono tutti possedere elementi dello stesso tipo. Per permettere stack con elementi di tipo diverso dobbiamo fare delle copie del file header e di quello sorgente dello stack ADT, oltre che modificare alcuni di questi file in modo che il tipo `Stack` e le funzioni a lui associate abbiano nomi diversi.

Quello che vorremmo avere è un singolo tipo stack "generico" dal quale poter creare uno stack di interi, di stringhe o di ogni altro tipo di cui potremmo aver bisogno. In C ci sono diversi modi per creare un tipo di questo genere, sebbene nessuno sia pienamente soddisfacente. L'approccio più comune utilizza `void *` come tipo degli elementi, il quale permette di inserire e prelevare puntatori di tipo arbitrario. Con questa tecnica il file `stackADT.h` sarebbe simile alla nostra versione originale, anche se i prototipi delle funzioni `push` e `pop` si presenterebbero in questo modo:

```
void push(Stack s, void *p);
void *pop(Stack s);
```

la funzione `pop` restituisce un puntatore all'elemento che viene prelevato dallo stack. Se lo stack è vuoto la funzione restituisce un puntatore nullo.

Nell'utilizzare `void *` come tipo degli elementi ci sono due svantaggi. Il primo è che questo approccio non funziona con dati che non possono essere rappresentati sotto forma di puntatore. Gli elementi possono essere delle stringhe (che sono rappresentate da un puntatore al primo carattere della stringa) o strutture allocate dinamicamente.

micamente ma non tipi base come `int` e `double`. Il secondo svantaggio sta nel fatto che il controllo degli errori non è più possibile. Uno stack che salvi elementi `void *` ammetterà facilmente un miscuglio di puntatori a tipi differenti. Non c'è modo per rilevare un errore causato dall'inserimento di un puntatore del tipo sbagliato.

ADT nei linguaggi più recenti

I problemi di cui abbiamo appena discusso sono trattati in modo molto più "pulito" nei linguaggi basati sul C più recenti come il C++, Java e il C#. Le collisioni tra i nomi vengono evitate definendo i nomi delle funzioni all'interno di una classe. Uno stack ADT verrebbe rappresentato da una classe `Stack`. Le funzioni dello stack apparterrebbero a quella classe e verrebbero riconosciute dal compilatore solo quando applicate all'oggetto `Stack`. Questi linguaggi possiedono una funzionalità chiamata di gestione delle eccezioni che permette alle funzioni come `push` e `pop` di "lanciare" un'eccezione quando viene rilevata una condizione di errore. Il codice dei client può gestire questo errore "afferrando" l'eccezione. I linguaggi C++, Java e C# forniscono anche delle speciali caratteristiche per definire gli ADT. In C++, per esempio, possiamo definire un template lasciando il tipo degli elementi non specificato.

Domande & Risposte

D: Il C non è stato pensato per scrivere programmi di grandi dimensioni. UNIX non è forse un programma di grandi dimensioni? [p. 500]

R: Non al tempo in cui il C è stato progettato. In un articolo del 1978, Ken Thompson stimò che il kernel di UNIX fosse costituito all'incirca da 10.000 righe di codice (più una piccola quantità di assembler). Le altre componenti di UNIX erano di dimensioni confrontabili. In un altro articolo del 1978, Dennis Ritchie e i suoi colleghi affermano che le dimensioni del compilatore C di un PDP-11 fossero di 9660 righe di codice. Per gli standard attuali questi sono programmi piccoli.

D: Ci sono alcuni tipi di dato astratti nella libreria del C?

R: Tecnicamente no, ma alcuni si avvicinano ad esserlo, tra cui il tipo `FILE` [tipo `FILE` > 22.1] (definito in `<stdio.h>`). Prima di poter eseguire un'operazione su un file, dobbiamo dichiarare una variabile di tipo `FILE` *:

`FILE *fp;`

La variabile `fp` verrà poi passata alle varie funzioni per la manipolazione dei file.

Ai programmatore viene chiesto di trattare `FILE` come un'astrazione. Per usare il tipo `FILE`, non è necessario conoscere cosa sia. Presumibilmente `FILE` è un tipo struttura ma lo standard C non lo garantisce. Infatti è meglio non sapere troppo su come i valori `FILE` vengano memorizzati dato che la definizione di questo tipo può variare da un compilatore all'altro (e spesso lo fa).

Naturalmente possiamo sempre andare nel file `stdio.h` e guardare cosa sia il tipo `FILE`. Dopo averlo fatto non c'è nulla che ci impedisca dallo scrivere codice che acceda all'intero di `FILE`. Per esempio, potremmo scoprire che `FILE` è una struttura con un membro chiamato `bsize` (la dimensione del buffer del file):

```
typedef struct {
    int bsize; /* dimensione del buffer */
} FILE;
```

Una volta che siamo a conoscenza dell'esistenza del membro `bsize`, non c'è nulla che ci impedisca di accedere alla dimensione del buffer di un particolare file:

```
printf("Buffer size: %d\n", fp->bsize);
```

Tuttavia farlo non è una buona idea perché altri compilatori C potrebbero salvare la dimensione del buffer del file con un nome diverso, o tenere traccia di questa in un modo completamente diverso. Modificare il membro `bsize` è un'idea persino peggiore:

```
fp->bsize = 1024;
```

A meno di non conoscere tutti i dettagli su come vengono memorizzati i file, questa è una cosa pericolosa da fare. Anche se conosciamo tutti i dettagli, questi possono cambiare con un diverso compilatore o con una versione differente dello stesso compilatore.

D: Oltre i tipi struttura incompleti che altri tipi incompleti sono presenti? [p. 508]

R: Uno dei tipi incompleti più comuni lo si incontra quando un vettore viene dichiarato senza specificare la sua dimensione:

```
extern int a[];
```

Dopo questa dichiarazione (che abbiamo incontrato per la prima volta nella Sezione 15.2), `a` è di un tipo incompleto in quanto il compilatore non ne conosce la lunghezza. Si presume che la variabile `a` venga definita in un altro file del programma. Quella definizione fornirà la lunghezza. Un altro tipo incompleto viene incontrato nelle dichiarazioni che non specificano la lunghezza per un vettore ma ne forniscono un inizializzatore:

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

In questo esempio il vettore `a` è inizialmente di tipo incompleto, tuttavia il tipo viene "completato" dall'inizializzatore.

Anche dichiarare il tag di un'unione senza specificare i suoi membri genera un tipo incompleto. I membri vettore flessibili [**membri vettore flessibili > 17.9**] (una caratteristica del C99) sono di tipo incompleto. Infine anche `void` è un tipo incompleto. Il tipo `void` ha l'insolita proprietà di non essere mai "completabile", il che rende impossibile la dichiarazione di una variabile di questo tipo.

D: Che altre restrizioni ci sono nell'utilizzo dei tipi incompleti? [p. 508]

R: L'operatore `sizeof` non può essere applicato su un tipo incompleto (questo non è sorprendente visto che la dimensione di un tipo incompleto è sconosciuta). Un membro di una struttura o di un'unione (a parte i membri vettore flessibili) non può essere di tipo incompleto. Analogamente neanche gli elementi di un vettore

possono essere di tipo incompleto. Infine neanche un parametro di una funzione può essere di tipo incompleto (sebbene questo sia ammesso nella dichiarazione della funzione). Il compilatore "regola" ogni parametro vettore presente nella definizione di una funzione in modo che sia di tipo puntatore, evitando così che questo sia di tipo incompleto.

Esercizi

Sezione 19.1

- Una coda (*queue*) è simile a uno stack ma differisce da questo per il fatto che gli elementi vengono aggiunti a un capo ma rimossi dall'altro secondo una modalità detta **FIFO** (first-in, first-out). Le operazioni su una coda includono:

- inserimento di un elemento alla fine della coda;
- rimozione di un elemento dall'inizio della coda;
- restituzione del primo elemento della coda (senza modificare la coda stessa);
- restituzione dell'ultimo elemento della coda (senza modificare la coda stessa);
- controllare se la coda è vuota.

Scrivete un'interfaccia per un modulo coda sotto forma di un file header chiamato *queue.h*.

Sezione 19.2

- Modificate il file *stack2.c* in modo da utilizzare le macro **PUBLIC** e **PRIVATE**.
- (a) Scrivete un'implementazione del modulo coda descritto nell'Esercizio 1 che sia basata su un vettore. Utilizzate tre interi per tenere traccia dello stato della coda. Il primo intero memorizzerà la posizione del primo slot libero all'interno del vettore (che viene utilizzato quando viene inserito un elemento). Il secondo intero memorizzerà la posizione del prossimo elemento che deve essere rimosso. Il terzo intero conterrà il numero di elementi presenti nella coda. Un inserimento o una rimozione che causasse l'incremento oltre la fine del vettore di uno dei primi due interi, dovrà invece riportare la variabile al valore zero facendo sì che questa riparta dall'inizio del vettore stesso.
- (b) Scrivete un'implementazione basata su una lista concatenata per il modulo coda descritto nell'Esercizio 1. Utilizzate due puntatori, uno che punti al primo nodo della lista e l'altro che punti all'ultimo nodo. Quando nella coda viene inserito un elemento, aggiungetelo alla fine della lista. Quando dalla coda viene rimosso un elemento, eliminate il primo nodo della lista.

Sezione 19.3

- (a) Scrivete un'implementazione del tipo **Stack** assumendo che **Stack** sia una struttura contenente un vettore di lunghezza prefissata.
- Ricreate il tipo **Stack**, utilizzando questa volta una rappresentazione basata su una lista concatenata invece che su un vettore (come riferimento guardate i file *stack.h* e *stack.c*).
- Modificate l'header *queue.h* dell'Esercizio 1 in modo che definisca il tipo **Queue**, dove **Queue** è una struttura contenente un vettore di lunghezza predeterminata (guardate l'Esercizio 3(a)). Modificate le funzioni presenti in *queue.h* in modo che accettino un parametro **Queue ***.

Sezione 19.4

6. (a) Aggiungete al file stackADT.c la funzione peek. Questa funzione dovrà avere un parametro di tipo Stack. Quando chiamata, la funzione restituisce l'elemento in cima allo stack senza modificare quest'ultimo.
(b) Ripetete il punto (a) modificando questa volta il file stackADT2.c.
(c) Ripetete il punto (a) modificando questa volta il file stackADT3.c.
7. Modificate il file stackADT2.c in modo che lo stack raddoppi automaticamente la propria dimensione in caso di riempimento. Fate in modo che la funzione push allochi dinamicamente il nuovo vettore. Questo deve presentare una dimensione doppia rispetto a quella del vettore usato precedentemente oltre che contenere una copia di tutti gli elementi. Assicuratevi che la funzione push deallochi il vecchio vettore una volta che i dati sono stati tutti copiati.

Progetti di programmazione

1. Modificate il Progetto di programmazione 1 del Capitolo 10 in modo che utilizzi lo stack ADT descritto nella Sezione 19.4. Potete utilizzare una qualsiasi delle implementazioni descritte in quella sezione.
2. Modificate il Progetto di programmazione 6 del Capitolo 10 in modo che utilizzi lo stack ADT descritto nella Sezione 19.4. Potete utilizzare una qualsiasi delle implementazioni descritte in quella sezione.
3. Modificate il file stackADT3.c della Sezione 19.4 aggiungendo alla struttura stack_type un membro di tipo int chiamato len. Questo membro terrà traccia del numero di elementi attualmente contenuti nello stack. Aggiungete anche una nuova funzione chiamata length che accetti un parametro Stack e restituisca il valore del membro len (dovranno essere modificate anche alcune delle funzioni già presenti nel file). Modificate il file stackclient.c in modo che chiami la funzione length (e visualizzi il valore restituito da questa) dopo ogni operazione che modifica lo stack.
4. Modificate i file stackADT.h e stackADT.c della Sezione 19.4 in modo che lo stack contenga valori di tipo void *, così come descritto nella Sezione 19.5. Il tipo Item non verrà più usato. Modificate il file stackclient.c in modo che salvi puntatori a stringhe all'interno degli stack s1 ed s2.
5. Partendo dall'header queue.h dell'Esercizio 1, create un file chiamato queueADT che definisca il seguente tipo Queue:

```
typedef struct queue_type *Queue;
```

queue_type è un tipo di struttura incompleto. Create un file chiamato queueADT.c che contenga una piena definizione di queue_type così come le definizioni di tutte le funzioni presenti in queue.h. Per immagazzinare gli elementi della coda utilizzate un vettore di lunghezza prefissata (guardate l'Esercizio 3). Create un file chiamato queueclient.c (simile al file stackclient.c della Sezione 19.4) che istanzi due code ed esegua delle operazioni su di esse. Non dimenticatevi di scrivere le funzioni create e destroy per la vostra struttura ADT.

6. Modificate il Progetto di programmazione 5 in modo che gli elementi presenti nella coda vengano salvati in un vettore allocato dinamicamente la cui lunghezza viene passata alla funzione `create`.
7. Modificate il Progetto di programmazione 5 in modo che gli elementi presenti in una coda vengano memorizzati in una lista concatenata (si veda l'Esercizio 3(b)).

20 Programmazione a basso livello

I capitoli precedenti hanno descritto le caratteristiche del C ad alto livello e indipendenti dalla macchina in uso. Sebbene queste caratteristiche siano adeguate per molte applicazioni, alcuni programmi hanno bisogno di eseguire delle operazioni a livello di bit. La manipolazione dei bit e le altre operazioni a basso livello sono particolarmente utili per scrivere programmi di sistema (che includono i compilatori e i sistemi operativi), programmi di cifratura, programmi di grafica e programmi per i quali sono importanti la velocità e/o l'uso efficiente della memoria.

La Sezione 20.1 tratta gli operatori *bitwise* del C i quali forniscono un modo semplice per accedere sia a particolari bit che a campi di bit. La Sezione 20.2 illustrerà la dichiarazione di strutture contenenti campi di bit. Infine la Sezione 20.4 descriverà come certe funzionalità ordinarie del C (definizione di tipi, le unioni e i puntatori) possano facilitare la scrittura di programmi a basso livello.

Alcune delle tecniche descritte in questo capitolo dipendono dalla conoscenza di come i dati vengono mantenuti nella memoria, il che può variare a seconda della macchina e del compilatore in uso. Fare affidamento a queste tecniche molto probabilmente renderà il programma non portabile, di conseguenza è meglio evitarle a meno che non siano assolutamente necessarie. Nel caso ne aveste bisogno, cercate di limitare il loro utilizzo solo a certi moduli del vostro programma, non diffondeteli e, cosa più importante, assicuratevi di documentare tutto quello che fate.

20.1 Operatori bitwise

Il C fornisce sei **operatori bitwise** che operano a livello di bit su dati di tipo intero. Per prima cosa tratteremo i due operatori di scorrimento, successivamente ci focalizzeremo sugli operatori bitwise rimanenti (operatore complemento bitwise, *and* bitwise, *or* esclusivo bitwise e *or* inclusivo bitwise).

Operatori di scorrimento bitwise

Gli operatori di scorrimento bitwise possono trasformare la rappresentazione binaria di un intero facendo scorrere i suoi bit verso sinistra o verso destra. Il C fornisce a tale scopo due operatori, che sono visualizzati nella Tabella 20.1.

Tabella 20.1 Operatori di scorrimento bitwise

Simbolo	Significato
<<	scorrimento a sinistra
>>	scorrimento a destra

Gli operandi degli operatori << e >> possono essere di qualsiasi tipo intero (incluso char). Le promozioni intere avvengono su entrambi gli operandi e il risultato è del tipo assunto dall'operando sinistro dopo la promozione.

Il valore di $i << j$ viene ottenuto facendo scorrere di j posizioni verso sinistra i bit di i . Per ogni bit che "fuoriesce" dall'estremo sinistro di i viene aggiunto uno zero sul lato destro. Il valore di $i >> j$ viene ottenuto facendo scorrere di j posizioni verso destra i bit di i . Se i è di un tipo senza segno oppure possiede un valore non negativo, allora alla sua sinistra vengono aggiunti gli zeri necessari. Nel caso in cui i sia un numero negativo, il risultato dipende dall'implementazione. Alcune implementazioni aggiungono degli zeri nell'estremo sinistro, mentre altre preservano il bit di segno aggiungendo degli uno.

PORATIBILITÀ

Per la portabilità è meglio eseguire le operazioni di scorrimento solo su numeri senza segno.

Gli esempi seguenti illustrano l'effetto ottenuto applicando gli operatori di scorrimento sul numero 13 (per semplicità questi esempi, come gli altri all'interno di questa sezione, utilizzano degli interi di tipo short che tipicamente sono di 16 bit).

```
unsigned short i, j;
```

```
i = 13; /* i adesso vale 13 (binario 000000000001101) */
j = i << 2; /* j adesso vale 52 (binario 0000000000110100) */
j = i >> 2; /* j adesso vale 3 (binario 000000000000011) */
```

Così come illustrano questi esempi, nessuno dei due operatori modifica i suoi operandi. Per modificare una variabile facendo scorrere i suoi bit, dobbiamo usare gli operatori composti di assegnamento < $=$ e > $=$:

```
i = 13; /* i adesso vale 13 (binario 000000000001101) */
i <<= 2; /* i adesso vale 52 (binario 0000000000110100) */
i >>= 2; /* i adesso vale 13 (binario 00000000000001101) */
```



Gli operatori di scorrimento bitweise hanno precedenza inferiore rispetto agli operatori aritmetici e questo può causare delle sorprese. Per esempio, $i << 2 + 1$ significa $i << (2 + 1)$ e non $(i << 2) + 1$.

Altri operatori bitweise

La Tabella 20.2 elenca gli operatori bitweise rimanenti.

Tabella 20.2 Altri operatori bitwise

Simbolo	Significato
<code>~</code>	complemento bitwise
<code>&</code>	<i>and</i> bitwise
<code>^</code>	<i>or esclusivo</i> bitwise
<code> </code>	<i>or inclusivo</i> bitwise

L'operatore `~` è unario e sul suo operando vengono eseguite le promozioni intere. Gli altri operatori sono binari e sui loro operandi vengono eseguite le normali conversioni aritmetiche.

Gli operatori `~, &, ^ e |` eseguono delle operazioni booleane su tutti i bit appartenenti ai loro operandi. L'operatore `~` produce il complemento del suo operando dove gli zeri sostituiscono gli uni e gli uni sostituiscono gli zeri. L'operatore `&` effettua l'operazione di *and* booleano su tutti i bit corrispondenti dei due operandi. Gli operatori `^` e `|` sono simili (entrambi effettuano l'operazione booleana *or* sui bit appartenenti ai loro operandi), tuttavia l'operatore `^` produce uno 0 se entrambi gli operandi possiedono un bit a 1, mentre in quel caso l'operatore `|` produce un 1.



Non confondete gli operatori *bitwise* `&` e `|` con gli operatori *logici* `&&` e `||`. A volte gli operatori bitwise producono lo stesso risultato degli operatori logici, ma non sono assolutamente equivalenti a questi ultimi.

Gli esempi seguenti illustrano l'effetto ottenuto applicando gli operatori `~, &, ^ e |`:

```
unsigned short i, j, k;
```

```
i = 21; /* i adesso vale 21 (binario 000000000010101) */
j = 56; /* j adesso vale 56 (binario 0000000000111000) */
k = ~i; /* k adesso vale 65514 (binario 111111111101010) */
k = i & j; /* k adesso vale 16 (binario 0000000000010000) */
k = i ^ j; /* k adesso vale 45 (binario 0000000000101101) */
k = i | j; /* k adesso vale 61 (binario 0000000000111101) */
```

Gli valori mostrati per l'operazione `~i` è basato sull'assunzione che il tipo `unsigned short` occupi 16 bit.

L'operatore `~` merita una menzione speciale dato che può essere utilizzato per rendere i programmi a basso livello più portabili. Supponete di aver bisogno di un intero i cui bit siano tutti a 1. La tecnica migliore è quella di scrivere `~0` che non dipende dal numero di bit presenti in un intero. Analogamente se avessimo bisogno di un intero con tutti i bit a 1 a eccezione degli ultimi cinque, potremmo scrivere `~0x1f`.

Ciascuno degli operatori `~, &, ^ e |` possiede un ordine di precedenza diverso:

Maggiore: `~`

`&`

`^`

Minore: `|`

Ne risulta la possibilità di combinare questi operatori senza la necessità di dover impiegare le parentesi. Per esempio, possiamo scrivere $i \& \sim j \mid k$ al posto di $(i \& (\sim j)) \mid k$ e $i \wedge j \wedge \sim k$ al posto di $i \wedge (j \wedge (\sim k))$. Naturalmente mettere le parentesi per evitare confusioni non fa male.



La precedenza degli operatori $\&$, \wedge e \mid è minore di quella degli operatori relazionali e di uguaglianza. Di conseguenza le istruzioni come la seguente non mostreranno l'effetto desiderato:

```
if (status & 0x4000 != 0) ...
```

Invece di testare se `status & 0x4000` è diverso da zero, questa istruzione calcolerà l'espressione `0x4000 != 0` (che ha valore 1) e poi controllerà se il valore di `status & 1` è diverso da zero.

Gli operatori composti di assegnamento $\&=$, $\wedge=$ e $\mid=$ corrispondono agli operatori $\&$, \wedge e \mid :

```
i = 21; /* i adesso vale 21 (binario 0000000000010101) */
j = 56; /* j adesso vale 56 (binario 0000000000111000) */
i &= j; /* i adesso vale 16 (binario 0000000000010000) */
i ^= j; /* i adesso vale 40 (binario 0000000000101000) */
i |= j; /* i adesso vale 56 (binario 0000000000111000) */
```

Utilizzare gli operatori bitwise per accedere ai bit

Quando si fa programmazione a basso livello, spesso vi è la necessità di salvare informazioni sotto forma di singoli bit o gruppi di bit. Nella programmazione grafica, per esempio, potremmo voler raggruppare due o più pixel in un singolo byte. Usando gli operatori bitwise possiamo estrarre o modificare i dati che sono stati memorizzati in un piccolo numero di bit.

Assumiamo che `i` sia una variabile a 16 bit di tipo `unsigned short`, vediamo come si possono eseguire su di essa le più comuni operazioni a singolo bit:

- **Settare un bit.** Supponete di voler impostare a uno il bit 4 della variabile `i` (assumeremo che il bit più a sinistra – il **bit più significativo** – sia il bit numero 15 mentre il bit meno significativo venga considerato il bit numero 0). Il modo più semplice per impostare a 1 il quarto bit di `i` è quello di eseguire un'operazione di `or` con la costante `0x0010` (una “maschera” che contiene un bit a 1 nella posizione numero 4):

```
i = 0x0000; /* i adesso vale 0000000000000000 */
i |= 0x0010; /* i adesso vale 0000000000010000 */
```

Più in generale, se la posizione del bit è contenuta nella variabile `j`, per creare la maschera possiamo usare un operatore di scorrimento:

```
i |= 1 << j; /* set del bit j */
```

Per esempio, se `j` ha valore 3, allora `1 << j` vale `0x0008`.

- **Azzerare un bit.** Per azzerare il bit numero 4 della variabile i utilizziamo una maschera con un bit a 0 nella posizione 4 e tutti i bit a 1 nelle altre posizioni:

```
i = 0x00ff; /* i adesso vale 0000000011111111 */
i &= ~0x0010; /* i adesso vale 0000000011101111 */
```

Utilizzando la stessa idea, possiamo scrivere facilmente un'istruzione che azzeri un bit la cui posizione è contenuta in una variabile:

```
i &= ~(1 << j); /* azzerà il bit j-esimo */
```

- **Controllare un bit.** La seguente istruzione if controlla se il bit 4 della variabile i è pari a 1:

```
if (i & 0x0010) /* controlla il bit 4 */
```

Per controllare se il bit j-esimo ha valore 1, possiamo usare la seguente istruzione:

```
if (i & 1 << j) /* controlla il bit j-esimo */
```

Spesso, per rendere più facili le operazioni sui bit si assegnano loro dei nomi. Per esempio, supponete di volere che i bit 0, 1 e 2 di un numero corrispondano rispettivamente ai colori blu, verde e rosso. Per prima cosa definiremo i nomi che rappresentano le tre posizioni dei bit:

```
#define BLUE 1
#define GREEN 2
#define RED 4
```

Settare, azzerare e controllare il bit BLUE viene fatto nei seguenti modi:

```
i |= BLUE; /* setta il bit BLUE */
i &= ~BLUE; /* azzerà il bit BLUE */
if (i & BLUE) /* controlla il bit BLUE */
```

In questo modo diventa semplice anche eseguire queste operazioni contemporaneamente su più bit:

```
i |= BLUE | GREEN; /* setta i bit BLUE e GREEN */
i &= ~(BLUE | GREEN); /* azzerà i bit BLUE e GREEN */
if (i & (BLUE | GREEN)) /* controlla i bit BLUE e GREEN */
```

L'istruzione if controlla che siano imposti a 1 sia il bit BLUE che il bit GREEN.

Usare gli operatori bitwise per accedere a campi di bit

Gestire un gruppo di diversi bit consecutivi (un **campo di bit**) è leggermente più complicato che lavorare su singoli bit. Ecco alcuni esempi delle operazioni più comuni sui campi di bit:

- **Modificare un campo di bit.** Modificare un campo di bit richiede un *and* bitwise (per azzerare il campo di bit), seguito da un *or* bitwise (per salvare i nuovi bit all'interno del campo). L'istruzione seguente illustra come salvare il valore binario 101 nei bit dal 4 al 6 della variabile i:

```
i = i & ~0x0070 | 0x0050; /* salva 101 nei bit 4-6 */
```

L'operatore `&` azzerà i bit 4-6 di `i`, successivamente l'operatore `|` impone a 1 i bit 6 e 4. Fate attenzione al fatto che `i |= 0x0050` non funzionerebbe sempre perché imporrebbe a 1 i bit 6 e 4 ma non modificherebbe il bit 5. Per generalizzare un poco questo esempio assumiamo che la variabile `j` contenga il valore che deve essere memorizzato nei bit dal 4 al 6 della variabile `i`. Avremo bisogno di far scorrere `j` nella posizione corretta prima di effettuare l'`or` bitwise:

```
i = (i & ~0x0070) | (j << 4); /* salva j nei bit 4-6 */
```

L'operatore `|` possiede una precedenza inferiore rispetto agli operatori `&` e `<<`, di conseguenza se lo volessimo potremmo eliminare le parentesi:

```
i = i & ~0x0070 | j << 4;
```

- **Recuperare il valore di un campo di bit.** Quando un campo di bit si trova all'estremo destro di un numero (i bit meno significativi) ricavare il suo valore è piuttosto semplice. Per esempio, la seguente istruzione ricava il valore dei bit dallo 0 al 2 della variabile `i`:

```
j = i & 0x0007; /* recupera i bit 0-2 */
```

Se il campo di bit non si trova nell'estremo destro di `i`, allora possiamo far scorrere il campo di bit fino a raggiungere la posizione corretta prima di estrarre con l'operatore `&`. Per esempio, per estrarre i bit dal 4 al 6 di `i`, possiamo usare la seguente istruzione:

```
j = (i >> 4) & 0x0007; /* recupera i bit 4-6 */
```

PROGRAMMA

Cifratura XOR

Uno dei metodi più semplici per cifrare dati è quello di applicare l'operazione di `or` esclusivo (XOR) tra ogni carattere e una chiave segreta. Supponete che la chiave sia il carattere `&`. Se facciamo lo XOR di questa chiave con il carattere `z`, allora come risultato otteniamo il carattere `\` (assumendo di usare il set di caratteri ASCII [**set di caratteri ASCII > Appendice D**]):

00100110 (codice ASCII per `&`)

XOR 01111010 (codice ASCII per `z`)

01011100 (codice ASCII per `\`)

Per decifrare il messaggio dobbiamo applicare il medesimo algoritmo. In altre parole, cifrando un messaggio già cifrato otteniamo il messaggio originale. Per esempio, se facessimo lo XOR del carattere `&` con il carattere `\` otterremmo il carattere originale `z`:

00100110 (codice ASCII per `&`)

XOR 01011100 (codice ASCII per `z`)

01111010 (codice ASCII per `\`)

Il programma seguente (`xor.c`) cifra un messaggio applicando l'operazione di XOR tra ogni carattere e la chiave `&`. Il messaggio originale può essere immesso dall'utente o letto da un file utilizzando il reindirizzamento dell'input. Il messaggio cifrato può essere visualizzato sullo schermo o salvato in un file utilizzando il reindirizzamento

dell'output [**reindirizzamento dell'input e dell'output > 22.1**]. Supponete Per esempio che il file msg contenga le seguenti righe:

Trust not him with your secrets, who, when left
alone in your room, turns over your papers.

--Johann Kaspar Lavater (1741-1801)

Per cifrare il file msg e salvare il messaggio cifrato all'interno del file newmsg useremo il comando seguente:

xor <msg>>newmsg

ora il file newmsg contiene le righe:

rTSUR HIR NOK QORN _IST UCETCRU, QNI, QNCH JC@R
GJIHC OH _IST TIIK, RSTHU IPCT _IST VGVCTU.
--LINGHH mGUVTG jGPGRCT (1741-1801)

Per recuperare il messaggio originale visualizzandolo sullo schermo useremo il comando

xor <newmsg>

Il nostro esempio ha mostrato che il programma non modificherà alcuni caratteri, tra cui quelli corrispondenti alle cifre. Effettuare lo XOR su questi caratteri produrrebbe dei caratteri di controllo invisibili che su alcuni sistemi operativi provocherebbero qualche problema. Nel Capitolo 22 vedremo come evitare problemi quando si leggono o scrivono file che contengono caratteri di controllo. Fino ad allora ci premureremo di usare la funzione isprint [**funzione isprint > 23.5**] per assicurarsi che entrambi i caratteri originali e quelli cifrati siano stampabili (ovvero non siano caratteri di controllo). Se uno dei due caratteri fallisce il test, il programma stamperà il carattere originale invece di quello cifrato.

Ecco il programma finito che, come potete vedere, è particolarmente breve:

```
xor.c /* Effettua la cifratura XOR */

#include <ctype.h>
#include <stdio.h>

#define KEY '&'

int main(void)
{
    int orig_char, new_char;

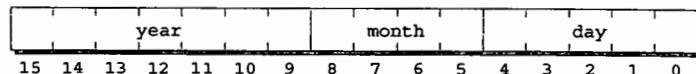
    while ((orig_char = getchar()) != EOF) {
        new_char = orig_char ^ KEY;
        if (isprint(orig_char) && isprint(new_char))
            putchar(new_char);
        else
            putchar(orig_char);
    }
    return 0;
}
```

20.2 Campi di bit nelle strutture

Sebbene le tecniche presentate nella Sezione 20.1 ci permettano di lavorare con i campi di bit, possono risultare scomode da utilizzare oltre che potenzialmente confuse. Fortunatamente il C fornisce un'alternativa: dichiarare delle strutture i cui membri rappresentano i campi di bit.

D&R

A titolo di esempio guardiamo a come il sistema operativo MS-DOS (spesso chiamato solamente DOS) salva la data di creazione e di ultima modifica di un file. Considerato che i giorni, i mesi e gli anni sono dei numeri piuttosto piccoli, salvati all'interno di normali interi sarebbe uno spreco di spazio. Per questo motivo il DOS alloca solamente 16 bit per una data. Al giorno sono associati 5 bit, 4 bit al mese e 7 bit all'anno:



Usando dei campi di bit possiamo definire una struttura C con una disposizione simile:

```
struct file_date {
    unsigned int day: 5;
    unsigned int month: 4;
    unsigned int year: 7;
};
```

Il numero posto dopo ogni membro indica la sua lunghezza espressa in bit. Dato che tutti i membri sono dello stesso tipo, possiamo anche condensare la dichiarazione:

```
struct file_date {
    unsigned int day: 5, month: 4, year: 7;
};
```

Il tipo di un campo di bit deve essere int, unsigned int oppure signed int.

Usare il tipo int è ambiguo dato che alcuni compilatori trattano il bit di ordine più alto del campo come un bit di segno mentre altri non lo fanno.

PORATIBILITÀ

C99

Dichiarare tutti i campi di bit come unsigned int oppure come signed int.

Nel C99 i campi di bit possono essere anche di tipo _Bool. I compilatori C99 possono anche permettere dei tipi aggiuntivi per i campi di bit.

Possiamo utilizzare un campo di bit esattamente come ogni altro membro di una struttura:

```
struct file_date fd;
fd.day = 28;
fd.month = 12;
fd.year = 8; /* rappresenta il 1988 */
```

Fate caso al fatto che il membro corrispondente all'anno, è rappresentato con riferimento al 1980 (cioè quello che secondo la Microsoft è l'anno di creazione del mondo). Dopo questi assegnamenti, la variabile fd si presenterà in questo modo:

0	0	0	1	0	0	0	1	1	0	0	1	1	1	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Avremmo potuto ottenere il medesimo risultato usando gli operatori bitwise, il che avrebbe reso il programma persino un po' più veloce. Tuttavia, scrivere un programma comprensibile di solito è più importante che guadagnare una manciata di millisecondi di tempo di esecuzione.

I campi di bit possiedono una restrizione che gli altri membri di una struttura non hanno. Dato che i campi di bit non possiedono un indirizzo nel suo senso comune, il C non ci permette di applicare su di essi l'operatore indirizzo (&). A causa di questa regola, funzioni come la scanf non possono salvare i dati direttamente all'interno di un campo di bit:

```
scanf("%d", &fd.day); /* SBAGLIATO */
```

Naturalmente possiamo sempre usare la scanf per salvare un dato in ingresso all'interno di una comune variabile e poi assegnarlo a fd.day.

Come vengono memorizzati i campi di bit

Vediamo ora come viene trattata dal compilatore la dichiarazione di una struttura contenente campi di bit come membri. Lo standard C accorda al compilatore un'ampia libertà nello scegliere come memorizzare i campi di bit.

Le regole concernenti il trattamento dei campi di bit si basano sul concetto di "unità di memorizzazione" (*storage units*). La dimensione di un'unità di memorizzazione è definita dall'implementazione. Valori tipici sono: 8 bit, 16 bit e 32 bit. Quando elabora la dichiarazione di una struttura, il compilatore raggruppa i campi di bit all'interno di un'unità di memorizzazione senza lasciare spazi tra i campi. Questo avviene fintanto che non c'è spazio a sufficienza per inserire il prossimo campo di bit, in tal caso alcuni compilatori saltano all'inizio della prossima unità di memorizzazione, mentre altri dividono il campo tra più unità di memorizzazione (quale dei due comportamenti venga seguito dipende dall'implementazione). Anche l'ordine nel quale i bit vengono disposti (da sinistra a destra o da destra a sinistra) dipende dall'implementazione.

Il nostro esempio file_date presume che le unità di memorizzazione siano lunghe 16 bit (un'unità di memorizzazione da 8 bit sarebbe comunque accettabile nel caso in cui il compilatore suddividesse il campo month tra due unità). Inoltre abbiamo assunto che i campi di bit siano stati disposti da destra a sinistra (con il primo campo che occupa i bit di ordine inferiore).

Il C ci permette di omettere il nome dei campi di bit. I campi senza nome sono utili come "riempimento" per assicurarsi che gli altri siano posizionati correttamente. Considerate l'orario associato a un file DOS, il quale viene salvato nel modo seguente:

```
struct file_time {
    unsigned int seconds: 5;
    unsigned int minutes: 6;
    unsigned int hours: 5;
};
```

Potreste chiedervi come sia possibile salvare i secondi (un numero che va da 0 a 59) in un campo di soli 5 bit. La risposta è che il DOS "imbroglia": divide il numero di secondi per 2, in questo modo il membro seconds contiene effettivamente un numero compreso tra 0 e 29. Se non siamo interessati al campo seconds possiamo togliere il suo nome:

```
struct file_time {
    unsigned int : 5;      /* inutilizzato */
    unsigned int minutes: 6;
    unsigned int hours: 5;
};
```

Gli altri campi di bit rimarranno allineati come se il campo seconds fosse ancora presente.

Un altro trucco che ci permette di controllare la disposizione dei campi di bit è quello di specificare la lunghezza di un campo senza nome pari a 0:

```
struct s {
    unsigned int a: 4;
    unsigned int : 0; /* campo di lunghezza 0 */
    unsigned int b: 8;
};
```

Un campo di lunghezza zero è un segnale per il compilatore di allineare i campi seguenti all'inizio di un'unità di memorizzazione. Se le unità di memorizzazione sono lunghe 8 bit, il compilatore allocherà 4 bit per il membro a, farà un salto di 4 bit fino all'unità successiva e poi allocherà 8 bit per il campo b. Se le unità di memorizzazione sono lunghe 16 bit, il compilatore allocherà 4 bit per a, salterà 12 bit e ne allocherà 8 per il membro b.

20.3 Altre tecniche a basso livello

Alcune caratteristiche del linguaggio di cui abbiamo discusso nei capitoli precedenti vengono usate spesso nella programmazione a basso livello. Per concludere questo capitolo daremo una scorsa a diversi esempi importanti: la definizione di tipi che rappresentino delle unità di memorizzazione, l'uso delle unioni per bypassare il normale controllo di tipo e l'utilizzo dei puntatori come indirizzi. Tratteremo anche il qualificatore volatile, che abbiamo evitato nella trattazione della Sezione 18.3 a causa della sua natura a basso livello.

Definire dei tipi indipendenti dalla macchina

Dato che il tipo `char` (per definizione) occupa un solo byte, delle volte tratteremo i caratteri come byte, usandoli per contenere dei dati che caratteri non sono. Quando lo facciamo è una buona pratica definire il tipo `BYTE`:

```
typedef unsigned char BYTE;
```

A seconda della macchina in uso potremmo voler definire dei tipi aggiuntivi. L'architettura x86 fa un uso estensivo delle word a 16 bit, di conseguenza una definizione come la seguente potrebbe rivelarsi utile per quella piattaforma:

```
typedef unsigned short WORD;
```

Nei prossimi esempi useremo i tipi `BYTE` e `WORD` appena definiti.

Usare le unioni per fornire diverse viste per i dati

Sebbene le unioni possano essere utilizzate in modo portabile (guardate la Sezione 16.4 per alcuni esempi), spesso nel C queste vengono utilizzate per uno scopo completamente differente: vedere un blocco di memoria in due o più modi diversi.

Ecco alcuni esempi basati sulla struttura `file_date` descritta nella Sezione 20.2. Dato che la struttura `file_date` occupa due byte, possiamo pensare un qualsiasi valore di due byte come una struttura `file_date`. In particolare, possiamo vedere un valore `unsigned char` come una struttura `file_date` (assumendo che gli interi `short` siano lunghi 16 bit). L'unione presentata di seguito ci permette di convertire facilmente un intero `short` nella data di un file e viceversa:

```
union int_date {
    unsigned short i;
    struct file_date fd;
};
```

Con l'aiuto di questa unione possiamo caricare da disco la data di un file sotto forma di due byte e poi estrarre i suoi campi `month`, `day` e `year`. Viceversa possiamo costruire una data sotto forma di struttura `file_date` e poi scriverla su disco sotto forma di una coppia di byte.

Come esempio di utilizzo dell'unione `int_date`, guardiamo una funzione che, quando le viene passato un argomento `unsigned short`, lo stampa sotto forma di data:

```
void print_date(unsigned short n)
{
    union int_date u;
    u.i = n;
    printf("%d/%d/%d\n", u.fd.month, u.fd.day, u.fd.year + 1980);
}
```

Usare le unioni per fornire viste multiple dei dati è particolarmente utile quando si lavora con i registri, che spesso sono divisi in unità più piccole. Nei processori x86, per esempio, troviamo dei registri chiamati `AX`, `BX`, `CX` e `. Ognuno di questi re-`

gistro può essere trattato come due registri da 8 bit. Il registro AX, per esempio, viene diviso nei registri AH e AL (le lettere H e L stanno per "high" e "low").

Quando scriviamo un'applicazione a basso livello per i computer basati sull'architettura x86, possiamo aver bisogno di variabili che rappresentino i registri AX, BX, CX e DX. Vogliamo accedere sia ai registri a 16 bit che a quelli a 8 bit, ma allo stesso tempo vogliamo mantenere la relazione esistente tra essi (modificare AX coinvolge sia AH che AL, e modificare AH o AL modifica di conseguenza anche AX). La soluzione è quella di creare due strutture: una contenente i membri che corrispondono ai registri a 16 bit, l'altra contenente i membri che corrispondono ai registri a 8 bit. Creiamo poi un'unione che racchiuda le due strutture:

```
union {
    struct {
        WORD ax, bx, cx, dx;
    } word;
    struct {
        BYTE al, ah, bl, bh, cl, ch, dl, dh;
    } byte;
} regs;
```

I membri della struttura word si sovrapporranno con i membri della struttura byte. Per esempio, ax occuperà la stessa memoria occupata da al e ah. Questo era esattamente quello che volevamo. Ecco un esempio che illustra come potrebbe essere usata l'unione regs:

```
regs.byte.ah = 0x12;
regs.byte.al = 0x34;
printf("AX: %hx\n", regs.word.ax);
```

modificare ah e al coinvolge anche ax, di conseguenza l'output sarà:

AX: 1234

Osservate che la struttura byte elenca al prima di ah anche se il registro AL corrisponde alla metà "inferiore" di AX e il registro AH a quella "superiore". La ragione è la seguente: quando un dato è composto da più di un byte, ci sono due modi per disporlo nella memoria, il primo è quello di disporre i byte nell'ordine "naturale" (con il byte più a sinistra disposto per primo) o con i byte nell'ordine inverso (il byte più a sinistra disposto per ultimo). La prima alternativa viene chiamata **big-endian**, mentre la seconda è conosciuta come **little-endian**. Il C non necessita di un particolare ordine per i byte visto che questo dipende dalla CPU sulla quale il programma verrà eseguito. Alcune CPU utilizzano l'approccio big-endian mentre altre usano quello little-endian. Cosa ha a che fare questo con la struttura byte? I processori x86 presumono che i dati siano memorizzati nell'ordine little-endian, di conseguenza il primo byte di regs.word.ax è di fatto il byte inferiore.

Normalmente non abbiamo bisogno di preoccuparci dell'ordinamento dei byte. Tuttavia i programmi che hanno a che fare con la memoria a basso livello devono preoccuparsi dell'ordine nel quale i byte sono disposti. L'ordinamento è importante anche quando lavoriamo con file che contengono dati che non sono caratteri.



Fate attenzione a quando utilizzate le unioni per fornire delle viste multiple dei dati. Dati che sono validi nel loro formato originale possono essere non validi se visti come un tipo diverso e questo potrebbe causare dei problemi imprevisti.

Usare i puntatori come indirizzi

Nella Sezione 11.1 abbiamo visto che, sebbene di solito non abbiamo bisogno di conoscerne i dettagli, un puntatore è effettivamente un qualche tipo di indirizzo di memoria. Quando si esegue la programmazione a basso livello, tuttavia, questi dettagli sono importanti.

Spesso un indirizzo è composto dallo stesso numero di bit che formano un intero (o intero di tipo long). Creare un puntatore che rappresenti un indirizzo specifico è semplice: possiamo semplicemente fare il cast di un intero in un puntatore. Di seguito viene illustrato come potremmo salvare l'indirizzo 1000 (esadecimale) in una variabile puntatore:

```
BYTE *p;  
p = (BYTE *) 0x1000; /* p contiene l'indirizzo 0x1000 */
```

PROGRAMMA

Visualizzare le locazioni di memoria

Il nostro prossimo programma permetterà all'utente di visualizzare segmenti di memoria del computer. Il programma si basa sulla disponibilità del C nel permettere che un intero venga usato come un puntatore. Tuttavia la maggior parte delle CPU eseguono i programmi in "modalità protetta", questo significa che un programma può accedere solo alle porzioni di memoria che gli appartengono. Questo previene il fatto che i programmi possano accedere o modificare la memoria appartenente ad altre applicazioni o allo stesso sistema operativo. Di conseguenza saremo in grado di utilizzare il nostro programma per visualizzare le sole aree di memoria che sono state allocate per l'uso del programma stesso. Andare al di fuori da queste aree provocherà il crash del programma.

Il programma viewmemory.c inizia visualizzando l'indirizzo della sua funzione main assieme a quello di una delle sue variabili. Questo fornirà all'utente un indizio di quali aree di memoria possono essere esaminate. Successivamente il programma chiederà all'utente di immettere un indirizzo (sotto forma di indirizzo esadecimale) e il numero di byte da visualizzare. Infine il programma visualizzerà un blocco di byte della lunghezza scelta a partire dall'indirizzo specificato.

I byte verranno visualizzati in gruppi di 10 (a eccezione dell'ultimo gruppo che può avere meno di 10 byte). L'indirizzo di un gruppo verrà visualizzato all'inizio della riga, lo seguiranno i byte del gruppo stesso (visualizzati sotto forma di numeri esadecimali) e successivamente gli stessi byte rappresentati come caratteri (dato che alcuni dei byte potrebbero rappresentare dei caratteri). Verranno visualizzati solo i byte stampabili (questo verrà determinato dalla funzione isprint), gli altri caratteri verranno visualizzati come dei punti.

Assumeremo che i valori int siano rappresentati da 32 bit e che anche gli indirizzi possiedano la medesima lunghezza. Com'è consuetudine gli indirizzi verranno visualizzati in formato esadecimale:

```
viewmemory.c /* Permette all'utente di visualizzare delle aree di memoria del computer */

#include <ctype.h>
#include <stdio.h>

typedef unsigned char BYTE;

int main(void)
{
    unsigned int addr;
    int i, n;
    BYTE *ptr;

    printf("Address of main function: %x\n", (unsigned int) main);
    printf("Address of addr variable: %x\n", (unsigned int) &addr);
    printf("\nEnter a (hex) address: ");
    scanf("%x", &addr);
    printf("Enter number of bytes to view: ");
    scanf("%d", &n);

    printf("\n");
    printf(" Address           Bytes           Characters\n");
    printf(" -----   -----   ----- \n");

    ptr = (BYTE *) addr;
    for (; n > 0; n -= 10) {
        printf("%8X ", (unsigned int) ptr);
        for (i = 0; i < 10 && i < n; i++)
            printf("%.2X ", *(ptr + i));
        for (; i < 10; i++)
            printf(" ");
        printf(" ");
        for (i = 0; i < 10 && i < n; i++) {
            BYTE ch = *(ptr + i);
            if (!isprint(ch))
                ch = '.';
            printf("%c", ch);
        }
        printf("\n");
        ptr += 10;
    }
    return 0;
}
```

Il programma è in qualche modo complicato dalla possibilità che il valore di n sia un multiplo di 10 e di conseguenza potrebbero esserci meno di 10 byte nell'ultimo gruppo. Due cicli for sono controllati dalla condizione $i < 10 \&\& i < n$. Qu

condizione fa sì che il ciclo venga eseguito 10 volte o n volte a seconda di quale sia il valore minore. Inoltre è presente anche un'istruzione per che compensa eventuali byte mancanti nell'ultimo gruppo stampando tre spazi per ogni byte mancante. In questo modo i caratteri che seguono l'ultimo gruppo di byte andrà ad allinearsi correttamente con i gruppi di caratteri delle righe precedenti.

La specifica di conversione %x utilizzata in questo programma è simile alla %x che era stata discussa nella Sezione 7.1. La differenza è che %x visualizza le cifre esadecimale A, B, C, D, E e F come lettere maiuscole, mentre la specifica %x le visualizza come lettere minuscole.

Ecco quello che potrebbe succedere compilando il programma con GCC e testandolo su un sistema x86 con sistema operativo Linux:

Address of main function: 804847c

Address of addr variable: bfef41154

Enter a (hex) address: 8048000

Enter number of bytes to view: 40

Address	Bytes	Characters
8048000	7F 45 4C 46 01 01 01 00 00 00	.ELF.....
804800A	00 00 00 00 00 00 02 00 03 00
8048014	01 00 00 00 C0 83 04 08 34 004.
804801E	00 00 C0 0A 00 00 00 00 00 00

Nell'esempio è stato chiesto al programma di stampare 40 byte a partire dall'indirizzo 8048000, il quale precede l'indirizzo della funzione main. Fate caso al byte 7F che viene seguito dai byte rappresentanti le lettere E, L e F. Questi quattro byte identificano il formato (ELF) nel quale il file eseguibile è stato salvato. Il formato ELF (*Executable and Linking Format*) è largamente utilizzato nei sistemi UNIX, Linux incluso. L'indirizzo 8048000 è l'indirizzo di default nel quale gli eseguibili ELF vengono caricati sulle piattaforme x86.

Facciamo girare ancora il programma, questa volta visualizzando un blocco di memoria che inizia dall'indirizzo della variabile addr:

Address of main function: 804847c

Address of addr variable: bfef5484

Enter a (hex) address: bfef5484

Enter number of bytes to view: 64

Address	Bytes	Characters
BFEC5484	84 54 EC BF B0 54 EC BF F4 6F	.T...T...o
BFEC548E	68 00 34 55 EC BF C0 54 EC BF	h.4U...T..
BFEC5498	08 55 EC BF E3 3D 57 00 00 00	.U....=W...
BFEC54A2	00 00 A0 BC 55 00 08 55 EC BFU..U..
BFEC54AC	E3 3D 57 00 01 00 00 00 34 55	=W.....4U
BFEC54B6	EC BF 3C 55 EC BF 56 11 55 00	..<U..V.U.
BFEC54C0	F4 6F 68 00	.oh.

Nessuno dei dati contenuti in quest'area della memoria è sotto forma di caratteri, conseguenza è un po' più complessa da decifrare. Tuttavia sappiamo una cosa: la variabile `addr` occupa i primi quattro byte di quest'area. Una volta presi in ordine inverso questi quattro byte formano il numero BFEC5484, ovvero l'indirizzo immesso dall'utente. Perché in ordine inverso? Perché, come abbiamo visto in precedenza in questa sezione, i processori x86 gestiscono i dati secondo l'ordinamento little-endian.

Il qualificatore di tipo volatile

Su alcuni computer certe locazioni di memoria sono "volatili", ovvero i valori contenuti in quelle locazioni che possono cambiare durante l'esecuzione del programma anche quando quest'ultimo non sta salvando nuovi valori al loro interno. Per esempio alcune locazioni di memoria possono contenere dei dati provenienti direttamente dai dispositivi di input.

Il qualificatore di tipo `volatile` ci permette di informare il compilatore nel caso cui dei dati utilizzati nel programma siano volatili. Tipicamente questo qualificatore compare nella dichiarazione di una variabile puntatore che punta a una locazione memoria di tipo `volatile`:

```
volatile BYTE *p; /* p punterà a un byte volatile */
```

Per capire perché il qualificatore `volatile` sia necessario, supponete che `p` punti a una locazione di memoria che contiene il più recente carattere digitato sulla tastiera dell'utente. Questa locazione è volatile: il suo valore cambia ogni volta che l'utente immette un carattere. Per ottenere i caratteri dalla tastiera e salvarli in un buffer potremmo utilizzare il ciclo seguente:

```
while (buffer non pieno) {
    attendi input;
    buffer[i] = *p;
    if (buffer[i++] == '\n')
        break;
}
```

Un compilatore sofisticato potrebbe accorgersi che questo ciclo non modifica né `*p` e quindi potrebbe ottimizzare il programma modificandolo e facendo in modo che `*p` venga caricato una volta soltanto:

```
salva *p in un registro;
while (buffer non pieno) {
    attendi input;
    buffer[i] = valore contenuto nel registro;
    if (buffer[i++] == '\n')
        break;
}
```

Il programma ottimizzato riempirebbe il buffer con tante copie dello stesso carattere (non è proprio quello che avevamo in mente). Dichiarare che `p` punta a dati volatili evita questo problema dicendo al compilatore che `*p` deve essere caricato dalla memoria ogni volta che viene utilizzato.

Domande & Risposte

D: Che cosa si intende dicendo che a volte gli operatori & e | producono lo stesso risultato degli operatori && e || ma che questo non accade sempre? [p. 527]

R: Confrontiamo i & j con i && j (osservazioni simili si applicano a | e ||). Fintanto che le variabili i e j contengono i valori 0 o 1 (in tutte le loro combinazioni), le due espressioni avranno sempre il medesimo valore. Tuttavia se i e j dovessero avere altri valori allora il risultato delle due espressioni potrebbe non combaciare sempre. Per esempio se i è uguale a 1 e j è uguale a 2, allora i & j avrà valore 0 (i e j non hanno bit corrispondenti a 1), mentre l'espressione i && j sarà pari a 1. Se i è uguale a 3 e j è uguale a 2, allora l'espressione i & j avrà il valore 2, mentre l'espressione i && j avrà il valore 1.

I side effect costituiscono un'altra differenza. Calcolare i & j++ incrementa *sempre* j come conseguenza di un side effect, mentre calcolare i && j++ incrementa j solo *delle volte*.

D: A chi interessa il modo in cui DOS salva le date? Il DOS non è "morto"? [p. 532]

R: Per la maggior parte sì. Tuttavia ci sono ancora molti file creati anni addietro le cui date sono memorizzate nel formato DOS. In ogni caso i file DOS sono un buon esempio per illustrare l'uso dei campi di bit.

D: Da dove provengono i termini "big-endian" e "little-endian"? [p. 536]

R: Ne *I viaggi di Gulliver* di Jonathan Swift, le isole immaginarie di Lilliput e Blefuscus sono costantemente in disaccordo su come aprire le uova sode, se aprirle dal lato più grande (*big end*) o dal lato più piccolo (*little end*). Naturalmente la scelta è arbitraria, proprio come il modo con cui ordinare i byte in un dato.

Esercizi

Sezione 20.1

1. *Mostrate l'output prodotto da ognuno dei seguenti frammenti di programma. Assumete che i, j e k siano variabili di tipo `unsigned short`.
 - (a) `i = 8; j = 9;`
`printf("%d", i >> 1 + j >> 1);`
 - (b) `i = 1;`
`printf("%d", i & ~i);`
 - (c) `i = 2; j = 1; k = 0;`
`printf("%d", ~i & j ^ k);`
 - (d) `i = 7; j = 8; k = 9;`
`printf("%d", i ^ j & k);`
2. Descrivete un modo semplice per effettuare su un bit il cosiddetto *toggle* (far passare il suo valore da 0 a 1 o da 1 a 0). Illustrate la tecnica scrivendo un'istruzione che effettui il toggle sul bit numero 4 della variabile i.

3. *Spiegate l'effetto che la macro seguente ha sui propri argomenti. Potete assumere che gli argomenti siano dello stesso tipo.

```
#define M(x,y) ((x)^=(y),(y)^=(x),(x)^=(y))
```

- W 4. Nella computer grafica, spesso i colori vengono memorizzati sotto forma di tre numeri rappresentanti le intensità di rosso, verde e blu. Supponete che ogni numero richieda otto bit e che si voglia salvare tutti e tre i valori in un singolo intero di tipo long. Scrivete una macro chiamata MK_COLOR avente tre parametri (le intensità di rosso, verde e blu). La macro dovrà restituire un valore di tipo long nel quale gli ultimi tre byte contengono le intensità di rosso, verde e blu. Il valore associato al rosso dovrà essere contenuto nell'ultimo byte mentre quello associato al verde dovrà essere contenuto nel penultimo byte.
5. Scrivete delle macro chiamate GET_RED, GET_GREEN e GET_BLUE tali che, dato un colore come argomento (si veda l'Esercizio 4), restituiscono le sue intensità su 8 bit di rosso, verde e blu.
- W 6. (a) Utilizzate gli operatori bitwise per scrivere la funzione seguente:

```
unsigned short swap_bytes(unsigned short i);
```

la funzione dovrà restituire il numero risultante dallo swap dei due byte di i (nella maggior parte dei computer gli interi short occupano due byte). Per esempio, se i possiede il valore 0x1234 (00010010 00110100 in binario), allora swap_byte dovrà restituire il valore 0x3412 (00110100 00010010 in binario). Testate la vostra funzione scrivendo un programma che legga un numero in esadecimale e poi lo riscriva dopo aver effettuato lo swap dei suoi byte:

Enter a hexadecimal number (up to four digits): 1234
Number with bytes swapped: 3412

Suggerimento: Per leggere e scrivere i numeri utilizzate la specifica di conversion %hx.

(b) Accorciate la funzione swap_bytes in modo che il suo corpo sia costituito da una singola istruzione.

7. Scrivete le seguenti funzioni:

```
unsigned int rotate_left(unsigned int i, int n);
unsigned int rotate_right(unsigned int i, int n);
```

la funzione rotate_left dovrà restituire il valore ottenuto facendo scorrere i bit di i di n posizioni verso sinistra. I bit che vengono "espulsi" dallo scorrimento devono essere spostati sul lato destro di i (per esempio: se gli interi sono lunghi 32 bit, allora la chiamata rotate_left(0x12345678, 4) dovrà restituire il valore 0x23456781). La funzione rotate_right è simile, ma dovrà "ruotare" i bit verso destra invece che verso sinistra.

- W 8. Sia f la seguente funzione:

```
unsigned int f(unsigned int i, int m, int n)
{
    return (i >> (m + 1 - n)) & ~(~0 << n);
}
```

(a) Qual è il valore di $\sim(0 \ll n)$?

(b) Cosa fa questa funzione?

9. (a) Scrivete la seguente funzione:

```
int count_ones(unsigned char ch);
```

la funzione dovrà restituire il numero di bit a 1 presenti in ch.

(b) Scrivete la funzione del punto (a) senza utilizzare un ciclo.

10. Scrivete la seguente funzione:

```
unsigned int reverse_bits(unsigned int n);
```

la funzione reverse_bits dovrà restituire un intero senza segno i cui bit sono gli stessi di quelli presenti in n ma in ordine inverso.

11. Ognuna delle seguenti macro definisce la posizione di un singolo bit all'interno di un intero:

```
#define SHIFT_BIT 1
#define CTRL_BIT 2
#define ALT_BIT 4
```

L'istruzione seguente è stata pensata per controllare se uno di questi bit è stato imposto al valore 1, tuttavia non visualizza mai il messaggio voluto. Spiegate perché l'istruzione non funziona a dovere e mostrate come correggerla. Assumete che key_code sia una variabile di tipo int.

```
if (key_code & (SHIFT_BIT | CTRL_BIT | ALT_BIT) == 0)
    printf("No modifier keys pressed\n");
```

12. La funzione seguente dovrebbe combinare due byte per formare un intero di tipo unsigned short. Spiegate perché la funzione non fa quanto voluto e mostrate come correggerla.

```
unsigned short create_short(unsigned char high_byte,
                           unsigned char low_byte)
{
    return high_byte << 8 + low_byte;
}
```

13. *Se n è una variabile di tipo unsigned int, che effetto avrà sui suoi bit l'istruzione seguente?

```
n &= n - 1;
```

Suggerimento: considerate l'effetto su n che si otterebbe eseguendo più di una volta l'istruzione.

14. Secondo lo standard IEEE per i numeri a virgola mobile, un valore di tipo float consiste di 1 bit di segno (il bit più significativo, ovvero quello che si trova più a sinistra), 8 bit di esponente e 23 bit di mantissa. Create una struttura che occupi 32 bit avente dei campi di bit corrispondenti al segno, all'esponente e alla man-

tissa. Dichiarate i campi di bit del tipo `unsigned int`. Controllate nel manuale del vostro compilatore per determinare l'ordine dei campi di bit.

15. *(a) Assumete che la variabile `s` sia stata dichiarata come segue:

```
struct {
    int flag: 1;
} s;
```

Con alcuni compilatori l'esecuzione delle istruzioni seguenti fa sì che venga visualizzato il valore 1, mentre con altri compilatori viene visualizzato -1. Spiegate le ragioni di questo comportamento.

```
s.flag = 1;
printf("%d\n", s.flag);
```

- (b) Come può essere evitato questo problema?

- Sezione 20.3** 16. A partire dal processore 386, le CPU x86 hanno dei registri a 32 bit chiamati EAX, EBX, ECX e EDX. La seconda metà (i bit meno significativi) di questi registri è rispettivamente uguale ad AX, BX, CX e DX. Modificate l'unione `regs` in modo che includa anche questi registri. L'unione dovrà essere creata in modo che modificare EAX cambi il valore di AX e modificare AX cambi il valore della seconda metà di EAX (gli altri registri dovranno comportarsi in modo simile). Nelle strutture word e byte avrete bisogno di aggiungere alcuni membri "fasulli" corrispondenti alle altre metà dei registri EAX, EBX, ECX e EDX. Dichiarate il tipo dei nuovi registri `DWORD` (*double word*) che deve essere definito come `unsigned long`. Non dimenticatevi che l'architettura x86 segue l'ordinamento little-endian.

Progetti di programmazione

1. Sviluppate un'unione che renda possibile visualizzare un valore a 32 bit sia come un float che come una struttura descritta nell'Esercizio 14. Scrivete un programma che salvi un 1 nel campo di segno della struttura, 128 nel campo dell'esponente e 0 nel campo della mantissa. Successivamente stampate il valore float contenuto nell'unione (se avete impostato correttamente i campi di bit il valore visualizzato deve essere -2.0).

21 La libreria standard

Nei capitoli precedenti abbiamo guardato alla libreria del C una parte alla volta, questo capitolo si concentra invece sulla libreria nel suo complesso. La Sezione 21.1 elenca le linee guida generali per l'uso della libreria e descrive anche un trucco presente in alcuni header della libreria: usare una macro per "nascondere" una funzione. La Sezione 21.2 presenta una panoramica di ogni header della libreria del C89. La Sezione 21.3 illustra i nuovi header presenti nella libreria del C99.

I capitoli successivi tratteranno dettagliatamente gli header della libreria, raggruppando gli header che sono in relazione. Gli header `<stddef.h>` e `<stdbool.h>` sono molto brevi, di conseguenza si è scelto di trattarli all'interno di questo capitolo (rispettivamente nelle Sezioni 21.4 e 21.5).

21.1 Usare la libreria

 La libreria standard del C89 è divisa in 15 parti; ogni parte è descritta da un header. Il C99 possiede nove header aggiuntivi, per un totale di 24 header (Tabella 21.1).

Tabella 21.1 Header della libreria standard

<code><assert.h></code>	<code><inttypes.h>^t</code>	<code><signal.h></code>	<code><stdlib.h></code>
<code><complex.h>^t</code>	<code><iso646.h>^t</code>	<code><stdarg.h></code>	<code><string.h></code>
<code><ctype.h></code>	<code><limits.h></code>	<code><stdbool.h>^t</code>	<code><tgmath.h>^t</code>
<code><errno.h></code>	<code><locale.h></code>	<code><stddef.h></code>	<code><time.h></code>
<code><fenv.h>^t</code>	<code><math.h></code>	<code><stdint.h>^t</code>	<code><wchar.h>^t</code>
<code><float.h></code>	<code><setjmp.h></code>	<code><stdio.h></code>	<code><wctype.h>^t</code>

^tsolo C99

La maggior parte dei compilatori è fornita di una libreria molto più estesa che invariabilmente presenta molti header che non compaiono nella Tabella 21.1. Naturalmente gli header aggiuntivi non sono standard e quindi non potete contare sul fatto che siano disponibili con altri compilatori. Spesso questi header forniscono delle funzioni che sono specifiche per un particolare computer o sistema operativo (questo spiega perché non sono standard). Possono, per esempio, fornire delle funzioni che

permettono un maggiore controllo sullo schermo e sulla tastiera. Sono comuni anche gli header che supportano la grafica o una interfaccia utente basata su finestre.

Gli header standard consistono principalmente di prototipi di funzioni, definizioni di tipi e macro. Se uno dei nostri file contiene una chiamata a una funzione dichiarata in un header o utilizza uno dei tipi o delle macro definite in questo header, allora dobbiamo includere quest'ultimo all'inizio del file. Quando un file include diversi header standard, l'ordine con cui si presentano le direttive #include non ha alcuna importanza. È possibile persino includere un header standard più di una volta.

Restrizioni sui nomi utilizzati nella libreria

Un qualunque file che includesse un header standard dovrebbe obbedire a un paio di regole. Per prima cosa non può utilizzare per altri scopi i nomi delle macro definite in quell'header. Se, per esempio, un file includesse <stdio.h>, non potrebbe riutilizzare il nome NULL dato che nell'header è già stata definita una macro con quel nome. Secondariamente i nomi di libreria con scope di file (in particolare i nomi typedef) non possono essere ridefiniti a livello di file. Di conseguenza, se un file include <stdio.h>, non può definire size_t come un identificatore con scope di file, visto che <stdio.h> definisce size_t come un nome typedef.

Sebbene queste restrizioni siano piuttosto ovvie, il C impone altre restrizioni che potreste non aspettarvi.

- **Gli identificatori che iniziano con il carattere underscore seguito da una lettera maiuscola o da un secondo carattere underscore** sono riservati per usi interni alla libreria. I programmi non devono mai utilizzare per nessun scopo dei nomi che seguono questo formato.
- **Gli identificatori che iniziano con il carattere underscore** sono riservati per essere usati come identificatori e tag con scope di file. Questi nomi non saranno mai utilizzati per i propri scopi a meno che non siano dichiarati all'interno di una funzione.
- **Ogni identificatore con collegamento esterno presente nella libreria standard** è riservato per l'uso come identificatore con collegamento interno. In particolare i nomi di tutte le funzioni della libreria standard sono riservati. Quindi, anche se un file *non* include <stdio.h>, non deve definire una funzione esterna chiamata printf dato che nella libreria c'è già una funzione con questo nome.

Queste regole si applicano a *tutti* i file di un programma, indipendentemente da quando i header vengano inclusi da tale file. Sebbene queste regole non vengano sempre attuite, non seguirle può compromettere la portabilità del programma.

Le regole sopra elencate non si applicano solo ai nomi che sono utilizzati attualmente dalla libreria, ma anche ai nomi che sono riservati per utilizzi futuri. La descrizione completa di quali nomi siano riservati è piuttosto lunga, la troverete nelle standard C sotto il nome *future library directions*. Per fare un esempio, il C riserva dei nomi che iniziano per str seguito da una lettera minuscola, visto che nomi di questo tipo possono essere aggiunti nell'header <string.h>.

Funzioni nascoste da macro

Per i programmatore C è comune sostituire piccole funzioni con macro parametriche. Questa pratica viene seguita anche nella libreria standard. Lo standard C permette agli header di definire delle macro aventi lo stesso nome delle funzioni di libreria, tuttavia protegge il programmatore richiedendo che siano disponibili anche delle vere funzioni. Di conseguenza non è inusuale per un header di libreria dichiarare una funzione e definire una macro con lo stesso nome.

Abbiamo già visto un esempio di una macro che duplica una funzione di libreria. La getchar è una funzione di libreria dichiarata all'interno dell'header <stdio.h> che presenta il seguente prototipo:

```
int getchar(void);
```

soltanamente l'header <stdio.h> definisce la getchar anche come una macro:

```
#define getchar() getc(stdin)
```

Per default una chiamata alla getchar verrà trattata come un'invocazione alla macro (dato che i nomi della macro vengono sostituiti durante la fase di preprocessing).

La maggior parte delle volte saremo lieti di utilizzare una macro al posto della funzione vera e propria perché, probabilmente, renderà più veloce il nostro programma. Occasionalmente però, vorremo utilizzare una vera funzione, magari per minimizzare la dimensione del codice eseguibile.

Se questa necessità si presentasse, potremmo rimuovere la definizione della macro (guadagnando così l'accesso alla vera funzione) utilizzando la diretta #undef [direttiva #undef > 14.3]. Per esempio, potremmo rimuovere la definizione della macro getchar dopo l'inclusione di <stdio.h>:

```
#include <stdio.h>
#undef getchar
```

Nel caso in cui la getchar non fosse una macro non si verificherebbe alcun problema. La diretta #undef non ha alcun effetto quando le viene passato un nome che non è definito come una macro.

Come alternativa possiamo disabilitare singoli utilizzi di una macro inserendo parentesi tonde:

```
ch = (getchar)(); /* invece di ch = getchar(); */
```

Il preprocessore non può individuare una macro parametrica a meno che il suo nome non sia seguito da una parentesi tonda sinistra. Il compilatore non viene ingannato così facilmente, infatti può ancora riconoscere getchar come una funzione.

21.2 Panoramica della libreria C89

Ora faremo una breve panoramica degli header che compongono la libreria standard del C89. Questa sezione funge da "mappa" per poter capire facilmente di quale parte della libreria avete bisogno. Ogni header viene descritto più avanti in questo capitolo o nei capitoli seguenti.

<assert.h> Diagnostica

Contiene solo la macro `assert`, la quale ci permette di inserire dei controlli di auto-diagnosi all'interno del nostro programma. Se uno di questi controlli ha esito negativo il programma termina. [header <assert.h> > 24.1]

<ctype.h> Gestione dei caratteri

Provvede alle funzioni per la classificazione dei caratteri e per la conversione delle lettere da minuscole a maiuscole e viceversa. [header <ctype.h> > 23.5]

<errno.h> Errori

Fornisce `errno` ("error number"), un lvalue che può essere controllato dopo l'invocazione a certe funzioni di libreria per vedere se si è verificato un errore durante la chiamata. [header <errno.h> > 24.2]

<float.h> Caratteristiche dei tipi a virgola mobile

Contiene delle macro che descrivono le caratteristiche dei tipi a virgola mobile, inclusi il loro intervallo di valori e la loro accuratezza. [header <float.h> > 23.1]

<limits.h> Dimensione dei tipi interi

Contiene delle macro che descrivono le caratteristiche dei tipi interi (inclusi i tipi carattere), tra cui il massimo e il minimo valore rappresentabile. [header <limits.h> > 23.2]

<locale.h> Localizzazione

Contiene delle funzioni che permettono a un programma di adattare il suo comportamento a una particolare nazione o regione geografica. Il comportamento legato alla localizzazione include il modo in cui vengono stampati i numeri (quale carattere viene utilizzato come separatore decimale), il formato dei valori monetari (il simbolo della valuta, per esempio), il set di caratteri e la rappresentazione delle date e delle ore. [header <locale.h> > 25.1]

<math.h> Matematica

Provvede alle comuni funzioni matematiche, incluse quelle trigonometriche, iperboliche, esponenziali, logaritmiche, di elevamento a potenza, intero più prossimo, valore assoluto e resto. [header <math.h> > 23.3]

<setjmp.h> Salti non locali

Fornisce le funzioni `setjmp` e `longjmp`. La prima "segna" una posizione all'interno di un programma, mentre la seconda può essere usata per ritornare in quel punto in un secondo momento. Queste funzioni rendono possibile il salto da una funzione

un'altra ancora attiva, bypassando così il normale funzionamento di "ritorno" dalle funzioni. Le funzioni `setjmp` e `longjmp` vengono utilizzate principalmente per gestire seri problemi che potrebbero sorgere durante l'esecuzione del programma. [header <setjmp.h> 24.4]

<signal.h> Gestione dei segnali

Fornisce delle funzioni che gestiscono delle condizioni eccezionali (segnali), tra cui le interruzioni e gli errori di *run-time*. La funzione `signal` installa una funzione che deve essere chiamata nel caso in cui un dato segnale si verificasse. La funzione `raise` genera un segnale. [header <signal.h> 24.3]

<stdarg.h> Argomenti variabili

Fornisce dei mezzi per scrivere delle funzioni che, come la `printf` e la `scanf`, possono avere un numero di argomenti variabile. [header <stdarg.h> 26.1]

<stddef.h> Definizioni comuni

Contiene le definizioni dei tipi e delle macro usati più di frequente. [header <stddef.h> 21.4]

<stdio.h> Input/Output

Fornisce un vario assortimento di funzioni di input/output, incluse le operazioni sui file, sia quelli ad accesso sequenziale che quelli ad accesso casuale. [header <stdio.h> 22.1, 22.8]

<stdlib.h> Utilità generale

Questo è un header polivalente per le funzioni che non appartengono a nessuno degli altri header. Le funzioni in questo header possono convertire le stringhe in numeri, generare numeri pseudo-casuali, eseguire compiti di gestione della memoria, comunicare con il sistema operativo, fare ricerche e ordinamenti, eseguire conversioni tra caratteri multibyte e i wide-characters. [header <stdlib.h> 26.2]

<string.h> Gestione delle stringhe

Contiene delle funzioni che eseguono delle operazioni sulle stringhe, tra cui: la copia, il confronto, la ricerca, e anche funzioni che operano su arbitrari blocchi di memoria. [header <string.h> 23.6]

<time.h> Data e ore

Fornisce delle funzioni per determinare l'ora corrente (e la data), per manipolare le ore in generale e formattarle ai fini della visualizzazione. [header <time.h> 26.3]

21.3 Modifiche della libreria C99

Alcune delle più importanti modifiche del C99 hanno a che fare con la libreria standard. Queste modifiche ricadono all'interno di tre gruppi.

- **Header aggiuntivi.** La libreria dello standard C99 possiede nove header che non esistevano nel C89. Effettivamente tre di questi (`<iso646.h>`, `<wchar.h>` e `<wctype.h>`) sono stati aggiunti nel 1995, quando il C89 è stato revisionato. Gli altri sei header (`<complex.h>`, `<fenv.h>`, `<inttypes.h>`, `<stdbool.h>`, `<stdint.h>` e `<tgmath.h>`) sono nuovi nel C99.
- **Macro e funzioni aggiuntive.** Lo standard C99 aggiunge delle macro e delle funzioni a diversi header esistenti, principalmente a `<float.h>`, `<math.h>` e `<stdio.h>`. Le aggiunte all'header `<math.h>` sono così consistenti che le tratteremo in una sezione separata (Sezione 23.4).
- **Versione migliorata delle funzioni esistenti.** Nel C99 alcune delle funzioni esistenti, tra cui la `printf` e la `scanf`, possiedono delle funzionalità aggiuntive.

<complex.h>

Aritmetica complessa

Definisce le macro `complex` e `I` che sono utili quando si opera con i numeri complessi. Fornisce anche le funzioni per eseguire le operazioni matematiche sui numeri complessi. [header `<complex.h>` > 27.4]

<fenv.h>

Ambiente in virgola mobile

Dà accesso ai flag di stato e ai modi di controllo per i numeri a virgola mobile. Per esempio, un programma può controllare un flag per vedere se si è verificato un overflow durante un'operazione floating point o impostare un modo di controllo che specifichi di che entità debba essere l'arrotondamento. [header `<fenv.h>` > 27.6]

<inttypes.h>

Conversione di formato per i tipi interi

Definisce delle macro che possono essere utilizzate nelle stringhe di format per i tipi interi dichiarati in `<stdint.h>`. Prevede anche delle funzioni per lavorare con gli interi della grandezza maggiore. [header `<inttypes.h>` > 27.2]

<iso646.h>

Ortografie alternative

Definisce delle macro che rappresentano certi operatori (quelli contenenti i caratteri `&`, `|`, `~`, `!` e `^`). Queste macro sono utili per scrivere programmi in un ambiente dove questi caratteri possono non far parte del set di caratteri locale. [header `<iso646.h>` > 25.3]

<stdbool.h>

Valori e tipo booleani

Definisce le macro `bool`, `true` e `false`; oltre che delle macro che possono essere utilizzate per controllare se le prime sono state definite. [header `<stdbool.h>` > 21.5]

<stdint.h> Tipi interi

Dichiara i tipi interi con specifiche dimensioni e definisce le macro relative (come le macro che specificano il valore massimo e quello minimo rappresentabile con ogni tipo). Definisce anche macro parametriche che costruiscono le costanti intere per i tipi specifici. [header <stdint.h> > 27.1]

<tgmath.h> Matematica per tipi generici

Nel C99 ci sono versioni multiple di molte funzioni matematiche presenti negli header <math.h> e <complex.h>. Le macro per "tipi generici" presenti in <tgmath.h> sono in grado di individuare il tipo degli argomenti che vengono passati ed effettuare una chiamata alla funzione appropriata tra quelle presenti in <math.h> e <complex.h>. [header <tgmath.h> > 27.5]

<wchar.h> Utilità per i caratteri estesi multibyte e per i wide-characters

Fornisce delle funzioni per l'input/output dei wide-character e per la manipolazione delle stringhe formate con questi tipi di caratteri. [header <wchar.h> > 25.5]

<wctype.h> Classificazione dei wide-character e utilità di mapping

Questo header rappresenta la versione per i wide-character dell'header ctype.h. Fornisce funzioni per la classificazione e la modifica dei wide-character. [header <wctype.h> > 25.6]

21.4 L'header <stddef.h>: definizioni comuni

L'header <stddef.h> provvede alle definizioni delle macro e dei tipi utilizzati più frequentemente e non dichiara nessuna funzione. I tipi sono:

- `ptrdiff_t`. Il tipo che risulta dalla sottrazione di due puntatori.
- `size_t`. Il tipo restituito dall'operatore `sizeof`.
- `wchar_t`. Un tipo sufficientemente grande da poter rappresentare tutti i possibili caratteri di tutte le localizzazioni supportate.

Tutte e tre sono nomi per tipi interi: `ptrdiff_t` deve essere un tipo con segno, mentre `size_t` deve essere senza segno. Per maggiori informazioni circa il tipo `wchar_t` leggete la Sezione 25.2.

L'header <stddef.h> definisce anche due macro. Una di queste è `NULL` che rappresenta il null pointer. L'altra macro, `offsetof`, richiede due argomenti: `type` (il tipo di una struttura) e `member-designator` (un membro della struttura).

La macro `offsetof` calcola il numero di byte compresi tra l'inizio di una struttura e il membro specificato.

Considerate la struttura di pagina seguente.

```
struct s {
    char a;
    int b[2];
    float c;
};
```

il valore di `offsetof(struct s, a)` deve essere 0. Il C garantisce che il primo membro di una struttura debba avere il medesimo indirizzo della struttura stessa. Non possiamo invece dire con sicurezza quale sarà l'offset dei membri `b` e `c`. È possibile che `offsetof(struct s, b)` sia uguale a 1 (visto che `a` è lungo un byte) e che `offsetof(struct s, c)` sia uguale a 9 (assumendo di avere interi a 32 bit). Tuttavia, alcuni compilatori lasciano dei "buchi" (dei byte non utilizzati) all'interno delle strutture (leggente la Sezione D&R del Capitolo 6) che possono avere effetto sul valore prodotto da `offsetof`. Per esempio: se un compilatore dovesse lasciare un buco di tre byte dopo `a`, allora l'offset di `b` e `c` sarebbe pari a 4 e a 12 rispettivamente. Questa è la "bellezza" della macro `offsetof`: produce l'offset corretto per qualsiasi compilatore, permettendoci di scrivere programmi portabili.

Ci sono diversi utilizzi di `offsetof`. Supponete per esempio di voler salvare al l'interno di un file i primi due numeri di una struttura `s`, ignorando il membro `c`. Invece di usare la funzione `fwrite` [funzione `fwrite` > 22.6] facendole scrivere `sizeof(struct s)` byte, che salverebbe l'intera struttura, le diremo di salvare solo `offsetof(struct s, c)` byte.

Un'ultima osservazione: alcuni dei tipi e delle macro definite in `<stddef.h>` compaiono anche in altri header (la macro `NULL`, per esempio, è definita anche in `<locale.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>` e `<time.h>`, oltre che nell'header C99 `<wchar.h>`). Di conseguenza ben pochi programmi avranno bisogno di includere `<stddef.h>`.

21.5 L'header `<stdbool.h>` (C99): valori e tipo booleani

L'header `<stdbool.h>` definisce quattro macro:

- `bool` (definito uguale a `_Bool`)
- `true` (definito uguale a 1)
- `false` (definito uguale a 0)
- `_bool_true_false_are_defined` (definito uguale a 1)

Abbiamo visto molti esempi di come possono essere utilizzate le macro `bool`, `true` e `false`. I possibili utilizzi della macro `_bool_true_false_are_defined` sono molto più limitati. Un programma può utilizzare una direttiva del preprocessore (come `#if` o `#ifdef`) per testare questa macro prima di cercare di definire una sua versione di `bool`, `true` e `false`.

Domande & Risposte

D: Abbiamo notato che viene usato il termine "header standard" invece di "file header standard". C'è qualche ragione per non utilizzare la parola "file"?

R: Sì. Secondo lo standard C, un "header standard" non deve essere necessariamente un file. Sebbene la maggior parte dei compilatori salvino effettivamente gli header come dei file, questi potrebbero essere incorporati nel compilatore stesso.

D: La Sezione 14.3 ha descritto alcuni svantaggi dell'uso delle macro parametriche al posto delle funzioni. Alla luce di questi problemi, non è pericoloso fornire una macro al fine di sostituire una funzione della libreria standard? [p. 547]

R: Secondo lo standard C, una macro parametrica che sostituisca una funzione di libreria deve essere "interamente protetta" da parentesi e deve inoltre calcolare i suoi argomenti esattamente una volta. Queste regole scongiurano la maggior parte dei problemi menzionati nella Sezione 14.3.

Esercizi

Sezione 21.1

- Individuate sul vostro sistema dove vengono mantenuti i file header. Trovate gli header non standard e determinate lo scopo di ognuno di essi.
- Avendo individuato i file header sul vostro sistema (leggente l'Esercizio 1), trovate un header standard nel quale una macro nasconde una funzione.
- Quando una macro nasconde una funzione, quale deve presentarsi prima nel file? La definizione della macro o il prototipo della funzione? Giustificate la vostra risposta.
- Fate un elenco di tutti gli identificatori riservati nella sezione *future library directions* dello standard C99. Distinguete tra gli identificatori che sono riservati solo quando viene incluso uno specifico header e quelli che sono riservati come nomi esterni.
- *La funzione `islower` che appartiene a `<ctype.h>`, controlla se un carattere corrisponde a una lettera minuscola. Perché, secondo lo standard C, la seguente versione di `islower` implementata come una macro non è ammissibile? (Potete assumere che il set di caratteri è quello ASCII).


```
#define islower(c) ((c) >= 'a' && (c) <= 'z')
```
- Tipicamente l'header `<ctype.h>` definisce la maggior parte delle sue funzioni anche sotto forma di macro. Queste macro si basano su un vettore statico che viene dichiarato in `<ctype.h>` ma che viene definito in un file separato. Di seguito viene presentata una porzione di un tipico header `<ctype.h>`. Utilizzate questo esempio per rispondere alle domande seguenti.
 - Perché i nomi delle macro (come `_UPPER`) e il vettore `_ctype` iniziano col carattere underscore?
 - Spiegate cosa conterrà il vettore `_ctype`. Assumendo che il set di caratteri sia quello ASCII, mostrate i valori degli elementi del vettore presenti alla posizioni 9 (carattere tab), 32 (carattere spazio), 65 (carattere A) e 94 (carattere ^). Leggete la Sezione 23.5 per una descrizione di quello che ogni macro deve restituire.

(c) Qual è il vantaggio dell'uso di un vettore per implementare queste macro?

```
#define _UPPER 0x01      /* lettera maiuscola */
#define _LOWER 0x02        /* lettera minuscola */
#define _DIGIT 0x04         /* cifra decimale */
#define _CONTROL 0x08       /* carattere controllo */
#define _PUNCT 0x10         /* carattere di punteggiatura */
#define _SPACE 0x20         /* carattere di spazio bianco */
#define _HEX 0x40            /* cifra esadecimale */
#define _BLANK 0x80          /* carattere di spazio */

#define isalnum(c)  (_ctype[c] & (_UPPER|_LOWER|_DIGIT))
#define isalpha(c)   (_ctype[c] & (_UPPER|_LOWER))
#define iscntrl(c)   (_ctype[c] & _CONTROL)
#define isdigit(c)   (_ctype[c] & _DIGIT)
#define isgraph(c)   (_ctype[c] &
                           (_PUNCT|_UPPER|_LOWER|_DIGIT))
#define islower(c)   (_ctype[c] & _LOWER)
#define isprint(c)   (_ctype[c] &
                           (_BLANK|_PUNCT|_UPPER|_LOWER|_DIGIT))
#define ispunct(c)   (_ctype[c] & _PUNCT)
#define isspace(c)   (_ctype[c] & _SPACE)
#define isupper(c)   (_ctype[c] & _UPPER)
#define isxdigit(c)  (_ctype[c] & (_DIGIT|_HEX))
```

- Sezione 21.2** 7. In quale header standard vi aspettate di trovare ognuno dei seguenti elementi?
- Ⓐ (a) Una funzione che determina il giorno della settimana corrente.
 - Ⓑ (b) Una funzione che controlla se un carattere corrisponde a una cifra.
 - Ⓒ (c) Una macro che fornisce il più grande numero unsigned int.
 - Ⓓ (d) Una funzione che arrotonda un numero a virgola mobile all'intero maggiore
più prossimo.
 - Ⓔ (e) Una macro che specifica il numero di bit presenti in un carattere.
 - Ⓕ (f) Una macro che specifica il numero di cifre significative in un valore double.
 - Ⓖ (g) Una funzione che cerca un particolare carattere all'interno di una stringa.
 - Ⓗ (h) Una funzione che apre un file per la lettura.

Progetti di programmazione

1. Scrivete un programma che dichiari la struttura s (guardate la Sezione 21.4) e stampi le dimensioni e gli offset dei membri a, b e c (usate sizeof per trovare le dimensioni e offsetof per ricavare gli offset). Fate in modo che il programma stampe anche la dimensione dell'intera struttura. Da queste informazioni determinate se la struttura contiene o meno dei buchi. Nel caso ne contenesse descrivetevi la posizione e la dimensione di ognuno.

22 Input/Output

La libreria di input/output del C è la parte più corposa della libreria standard. In ragione della sua importanza dedicheremo un intero capitolo all'header `<stdio.h>`, il depositario principale delle funzioni di input/output.

Stiamo utilizzando l'header `<stdio.h>` fin dal Capitolo 2 e abbiamo una certa esperienza nell'uso delle funzioni `printf`, `scanf`, `putchar`, `getchar`, `puts` e `gets`. Questo capitolo fornisce maggiori informazioni su queste sei funzioni e allo stesso tempo ne introduce diverse nuove, la maggior parte delle quali hanno a che fare con i file. Fortunatamente molte di queste funzioni sono in relazione stretta con le funzioni che già conosciamo. Per fare un esempio, la funzione `fprintf` è la "versione su file" della funzione `printf`.

Inizieremo il capitolo con una discussione riguardante alcuni argomenti di base: il concetto di *stream*, il tipo `FILE`, il reindirizzamento dell'input e dell'output e la differenza esistente tra i file testuali e quelli binari (Sezione 22.1). Successivamente la trattazione si rivolgerà verso le funzioni che sono state sviluppate per un uso specifico sui file, tra queste vedremo le funzioni per l'apertura e la chiusura di un file (Sezione 22.2). Dopo aver trattato la `printf`, la `scanf` e le funzioni relative all'input/output formattato (Sezione 22.3), ci occuperemo delle funzioni che leggono e scrivono dati non formattati:

- `getc`, `putc` e le funzioni a loro collegate, che leggono e scrivono un *carattere* alla volta (Sezione 22.4);
- `gets`, `puts` e le funzioni a loro collegate, che leggono e scrivono una *riga* per volta (Sezione 22.5);
- `fread` e `fwrite`, che leggono e scrivono dei *blocchi* di dati (Sezione 22.6).

Successivamente la Sezione 22.7 illustra come effettuare operazioni di accesso casuale sui file. Infine la Sezione 22.8 descrive le funzioni `sprintf`, `snprintf` e `sscanf`, varianti della `printf` e della `scanf` per la scrittura e la lettura da stringa.

Questo capitolo tratta tutte le funzioni presenti nell'header `<stdio.h>` tranne otto. Una di queste otto, la funzione `perror`, è strettamente collegata all'header `<errno.h>`, quindi la sua trattazione viene rimandata alla Sezione 24.2 che si occupa di tale header. La Sezione 26.1 tratta le funzioni rimanenti (`vfprintf`, `vprintf`, `vsprintf`, `vsnprintf`, `vf-`

`scanf`, `vscanf`, e `vsscanf`). Queste funzioni si basano sul tipo `va_list` che verrà introdotto in quella sezione.

C99

Nel C89 tutte le funzioni standard di input/output appartenevano all'header `<stdio.h>`. Questo non succede nel C99 dove alcune funzioni di I/O vengono dichiarate nell'header `<wchar.h>`. Le funzioni appartenenti a `<wchar.h>` gestiscono i wide character invece dei consueti caratteri, la buona notizia, però, è che la maggior parte di queste funzioni somiglia a quelle dell'header `<stdio.h>`. Le funzioni presenti in `<stdio.h>` che leggono o scrivono dati, sono conosciute come **funzioni per input/output dei byte**, le funzioni simili presenti in `<wchar.h>` sono chiamate **funzioni per input/output dei wide character**.

22.1 Stream

Nel linguaggio C il termine *stream* indica una qualsiasi sorgente di input o una qualsiasi destinazione per l'output. Molti piccoli programmi, come quelli scritti nei capitoli precedenti, ottengono tutto il loro input da uno stream (solitamente associato alla tastiera) e scrivono tutto il loro output in un altro stream (tipicamente associato allo schermo).

I programmi più corposi possono aver bisogno di stream aggiuntivi. Spesso questi stream rappresentano i file memorizzati su vari mezzi (come gli hard disk, i CD, DVD e le memorie flash), ma possono essere facilmente associati a tutti i dispositivi che non immagazzinano file: porte di rete, stampanti e così via. Ci concentreremo sui file dato che sono comuni e facili da capire (a volte potremo anche utilizzare il termine *file* quando sarebbe più corretto dire *stream*). Tenete presente tuttavia che molte delle funzioni presenti in `<stdio.h>` possono lavorare altrettanto bene con tutti gli stream e non solo con quelli che rappresentano dei file.

Puntatori a file

In un programma C l'accesso a uno stream avviene per mezzo di un **puntatore a file**, che è di tipo `FILE *` (il tipo `FILE` viene dichiarato all'interno di `<stdio.h>`). Certi stream vengono rappresentati da puntatori a file che possiedono dei nomi standard in ogni caso, quando ne abbiamo bisogno, possiamo dichiarare dei puntatori a file aggiuntivi. Per esempio, se un programma avesse bisogno di due stream aggiuntivi oltre a quelli standard, potrebbe contenere la seguente dichiarazione:

`FILE *fp1, *fp2;`

Un programma può dichiarare un numero qualsiasi di variabili di tipo `FILE *`, sebbene solitamente i sistemi operativi limitino il numero di stream che possono essere aperti contemporaneamente.

Stream standard e reindirizzamento

L'header `<stdio.h>` fornisce tre stream standard (Tabella 22.1). Questi stream sono pronti all'uso: non abbiamo bisogno di dichiararli, né di aprirli o chiuderli.

Tabella 22.1 Stream standard

Puntatore a file	Significato	Significato di default
stdin	standard input	tastiera
stdout	standard output	schermo
stderr	standard error	schermo

Le funzioni che abbiamo utilizzato nei capitoli precedenti (`printf`, `scanf`, `putchar`, `getchar`, `puts` e `gets`) ottenevano l'input da `stdin` e inviavano l'output a `stdout`. Per default, `stdin` rappresenta la tastiera, mentre `stdout` e `stderr` rappresentano lo schermo. Tuttavia molti sistemi operativi permettono che questi significati di default possano essere modificati attraverso un meccanismo chiamato **reindirizzamento**.

D&R

Tipicamente possiamo forzare un programma a ottenere il suo input da un file invece che dalla tastiera inserendo il nome del file nella riga di comando preceduto dal carattere <:

```
demo <in.dat
```

Questa tecnica, conosciuta come **reindirizzamento dell'input** fa in modo che lo stream `stdin` rappresenti un file (`in.dat` in questo caso) invece che la tastiera. Il bello del reindirizzamento è che il programma `demo` non si accorge che sta leggendo da `in.dat`, per quel che ne sa tutti i dati che ottiene da `stdin` sono stati immessi alla tastiera.

Il **reindirizzamento dell'output** è simile. Solitamente lo stream `stdout` viene reindirizzato mettendo il nome di un file sulla riga di comando e facendolo precedere dal carattere >:

```
demo >out.dat
```

Tutti i dati inviati a `stdout` invece di comparire sullo schermo verranno scritti nel file `out.dat`. Possiamo anche combinare il reindirizzamento dell'input e quello dell'output:

```
demo <in.dat >out.dat
```

I caratteri < e > non devono necessariamente essere adiacenti ai nomi dei file, inoltre l'ordine nel quale questi ultimi sono elencati non ha alcuna importanza, infatti gli esempi seguenti funzioneranno ugualmente:

```
demo < in.dat > out.dat
```

```
demo >out.dat <in.dat
```

Un problema con il reindirizzamento dell'output è che *qualsiasi* cosa venga scritta su `stdout` viene inserita all'interno del file. Se il programma va fuori strada e inizia a scrivere messaggi di errore, non ce ne accorgeremo fino a quando non andremo a leggere all'interno del file. Questa è la ragione dell'esistenza di `stderr`. Scrivendo i messaggi di errore su `stderr` invece che su `stdout`, garantiamo che questi messaggi compariranno sullo schermo anche quando `stdout` viene reindirizzato (tuttavia capita spesso che i sistemi operativi permettano di reindirizzare anche `stderr`).

File testuali e file binari

L'header `<stdio.h>` supporta due tipi di file: quelli testuali e quelli binari. I byte all'interno di un **file testuale** rappresentano dei caratteri, cosa che rende possibile a un essere umano leggere il file e modificarlo. Il codice sorgente di un programma C, per esempio, viene contenuto in un file testuale. In un **file binario** d'altro canto, i byte non rappresentano necessariamente dei caratteri. Gruppi di byte possono rappresentare altri tipi di dati, come numeri interi e a virgola mobile. Un programma eseguibile C viene contenuto in un file di questo tipo, potete rendervene conto facilmente cercando di leggere all'interno del suo contenuto.

I file testuali possiedono due caratteristiche che i file binari non hanno:

- **I file testuali sono suddivisi in righe.** Solitamente ogni riga di un file testuale termina con uno o due caratteri speciali. Il codice di questi caratteri dipende dal sistema operativo. In Windows il segnalatore di fine riga è costituito da un carattere *carriage-return* ('`\x0d`') seguito da un carattere *line-feed* ('`\x0a`'). In UNIX e nelle nuove versioni del sistema operativo Macintosh (Mac OS), il segnalatore di fine riga è costituito da un singolo carattere *line-feed*. Versioni più vecchie di Mac OS utilizzano un singolo carattere *carriage-return*.
- **I file testuali possono contenere uno speciale segnalatore di termine file (end-of-file).** Alcuni sistemi operativi permettono che uno speciale byte venga utilizzato come segnalatore per la fine del file. In Windows questo segnalatore corrisponde a '`\x1a`' (Ctrl-Z). Non è necessario che Ctrl-Z sia presente, ma nel caso lo fosse, rappresenterebbe la fine del file. Tutti i byte oltre a Ctrl-Z vengono ignorati. La convenzione del Ctrl-Z è un'eredità del DOS, il quale a sua volta l'ha ereditata dal CP/M, uno dei primi sistemi operativi per personal computer. La maggior parte dei sistemi operativi, UNIX incluso, non possiede uno speciale carattere end-of-file.

I file binari non sono divisi in righe. In un file binario non ci sono segnalatori a indicare il termine di una riga o del file: tutti i byte sono trattati allo stesso modo.

Quando scriviamo un dato all'interno di un file, dobbiamo considerare se salvarlo in forma testuale o binaria. Per capire la differenza, consideriamo come potremmo salvare all'interno di un file il numero 32767. Una possibilità sarebbe quella di scrivere il numero in forma testuale ovvero con i caratteri 3, 2, 7, 6 e 7. Se il set di caratteri è quello ASCII allora otterremo i seguenti byte:

00110011	00110010	00110111	00110110	00110111
3	2	7	6	7

L'altra possibilità è quella di salvare il numero in formato binario, il che richiederebbe come minimo due byte:

01111111	11111111
----------	----------

I byte verranno invertiti sui sistemi che utilizzano l'ordinamento little-endian [**ordinamento little-endian > 20.3**]. Come illustrato da questo esempio, spesso salvare i numeri in forma binaria ci permette di risparmiare spazio.

Quando stiamo scrivendo un programma che legge e scrive su un file, dobbiamo tener conto se il file sia testuale o binario. Un programma che visualizza sullo schermo il contenuto di un file, probabilmente assumerà che sia un file testuale. Un programma che effettua la copia di un file, d'altro canto, non potrà assumere che il file che deve essere copiato sia un file testuale. Se lo facesse i file binari contenenti il carattere end-of-file non verrebbero copiati completamente. Quando non possiamo dire con sicurezza se un file sia testuale o binario, la cosa più sicura è assumere che sia in forma binaria.

22.2 Operazioni sui file

La semplicità è una delle attrattive derivanti dal reindirizzamento dell'input e dell'output. Non c'è bisogno di aprire un file, chiuderlo o effettuare qualsiasi altra operazione esplicita riguardante i file. Sfortunatamente, per molte applicazioni, il reindirizzamento è troppo limitante. Quando un programma si basa sul reindirizzamento non ha alcun controllo sui suoi file, non conosce nemmeno i loro nomi. Peggio ancora, il reindirizzamento non è di aiuto se il programma ha bisogno di leggere due file o scrivere su due file allo stesso tempo.

Quando il reindirizzamento non è sufficiente, utilizziamo le operazioni sui file che vengono fornite dall'header <stdio.h>. In questa sezione esploreremo queste operazioni che includono: l'apertura di un file, la chiusura di un file, la modifica della modalità di gestione del buffer associato a un file, la cancellazione di un file e la modifica del nome di un file.

Aprire un file

```
FILE *fopen(const char * restrict filename,
            const char * restrict mode);
```

- fopen** Aprire un file per usarlo come uno stream richiede una chiamata alla funzione fopen. Il primo argomento della funzione è una stringa contenente il nome del file che deve essere aperto (il "nome del file" può contenere informazioni riguardanti la sua posizione, come il drive o il percorso). Il secondo argomento è una "stringa di modalità" che specifica quali operazioni abbiamo intenzione di compiere sul file. La stringa "r", per esempio, indica che i dati verranno letti dal file e che nulla vi sarà scritto all'interno.

Osservate come nel prototipo della funzione fopen la keyword restrict compaia due volte [keyword **restrict** > 17.8]. La keyword restrict, che appartiene al C99, indica che gli argomenti filename e mode devono puntare a stringhe che non condividono locazioni di memoria. Il prototipo del C89 per la funzione fopen non contiene questa keyword, ma per il resto è identico. La keyword restrict non ha alcun effetto sul comportamento della fopen, di conseguenza solitamente può essere ignorata. In questo e nei capitoli seguenti scriveremo questa keyword in *corsivo* per ricordarci che è una caratteristica propria del C99.



I programmati Windows devono fare attenzione quando il nome del file in una chiamata fopen include il carattere \ dato che il C tratta questo carattere come l'inizio di una sequenza di escape [sequenze di escape > 7.3]. La chiamata

```
fopen("c:\project\test1.dat", "r")
```

non andrà a buon fine perché il compilatore tratta \t come un carattere di escape (\p non è un carattere di escape valido, ma ne somiglia uno. Lo standard C stabilisce che il suo significato debba essere indefinito). Ci sono due modi per evitare il problema. Il primo consiste nell'utilizzare \\ al posto di \:

```
fopen("c:\\project\\test1.dat", "r")
```

L'altra tecnica è ancora più semplice: utilizzate il carattere / al posto del carattere \:

```
fopen("c:/project/test1.dat", "r")
```

Windows accetterà senza problemi il carattere / come separatore delle directory.

La funzione fopen restituisce un puntatore a file che il programma può salvare all'interno di una variabile per utilizzarlo per effettuare operazioni sul file. Ecco una tipica chiamata alla fopen, dove fp è una variabile del tipo FILE *:

```
fp = fopen("in.dat", "r"); /* apre in.dat in lettura */
```

Quando in un secondo momento il programma chiama una funzione di input per leggere dal file in.dat, le fornirà il puntatore fp come argomento.

Quando non può aprire un file, fopen restituisce un puntatore nullo. Può essere che il file non esista o che sia nel posto sbagliato o che la funzione non abbia il permesso per aprirlo.



Non assumete mai a priori che un file possa essere aperto: controllate sempre il valore restituito dalla fopen per assicurarvi che non sia un puntatore nullo.

Modalità di apertura

Quale stringa di modalità passeremo alla funzione fopen dipende non solo da quali operazioni abbiamo intenzione di fare sul file ma anche dal tipo dei dati contenuti al suo interno (testuali o binari). Per aprire un file testuale utilizzeremo una delle stringhe presenti nella Tabella 22.2.

Tabella 22.2 Modalità di apertura per i file testuali

Stringa	Significato
"r"	Apre il file in lettura
"w"	Apre il file in scrittura (non è necessario che il file esista)
"a"	Apre il file in accodamento (non è necessario che il file esista)
"r+"	Apre il file in lettura e scrittura, comincia dall'inizio del file
"w+"	Apre il file in lettura e scrittura (tronca il file se esiste)
"a+"	Apre il file in lettura e scrittura (accoda se il file esiste)



Quando utilizziamo fopen per aprire un file binario, dobbiamo includere nella stringa la lettera b. La Tabella 22.3 elenca le modalità di apertura per i file binari.

Dalle Tabelle 22.2 e 22.3 vediamo che <stdio.h> distingue tra la *scrittura* e l'*accodamento* dei dati. Quando i dati vengono scritti in un file, solitamente sovrascrivono i dati precedenti. Quando un file viene aperto in accodamento (*Appending*) invece, i dati scritti nel file vengono aggiunti alla fine, preservando il contenuto originale del file.

In ogni caso, delle regole speciali si applicano quando un file viene aperto sia in lettura che scrittura (le modalità che contengono il carattere +). Non possiamo passare dalla lettura alla scrittura senza prima chiamare una funzione di posizionamento, a meno che l'operazione di lettura non abbia incontrato la fine del file [funzione di posizionamento nei file > 22.7]. Inoltre non possiamo passare dalla scrittura alla lettura senza chiamare la funzione fflush (trattata successivamente in questa sezione) o chiamare una funzione di posizionamento.

Tabella 22.3 Modalità di apertura per i file binari

Stringa	Significato
"rb"	Apre il file in lettura
"wb"	Apre il file in scrittura (non è necessario che il file esista)
"ab"	Apre il file in accodamento (non è necessario che il file esista)
"r+b" oppure "rb+"	Apre il file in lettura e scrittura, comincia dall'inizio del file
"w+b" oppure "wb+"	Apre il file in lettura e scrittura (tronca il file se esiste)
"a+b" oppure "ab+"	Apre il file in lettura e scrittura (accoda se il file esiste)

Chiudere un file

```
int fclose(FILE *stream);
```

fclose La funzione fclose permette a un programma di chiudere un file che non viene più utilizzato. L'argomento di fclose deve essere un puntatore a file ottenuto da una chiamata alla fopen o alla freopen (trattata più avanti in questa sezione). La funzione fclose restituisce uno zero se il file è stato chiuso con successo, altrimenti restituisce il codice di errore EOF (una macro definita in <stdio.h>).



Per vedere come la fopen e la fclose vengano utilizzate nella pratica, di seguito viene presentato un programma che apre in lettura il file example.dat, controlla se sia stato aperto con successo e poi lo chiude prima del termine del programma stesso:

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

#define FILE_NAME "example.dat"

int main(void)
{
    FILE *fp;
```

```

fp = fopen(FILE_NAME, "r");
if (fp == NULL) {
    printf("Can't open %s\n", FILE_NAME);
    exit(EXIT_FAILURE);
}

fclose(fp);
return 0;
}

```

Naturalmente non è insolito trovare delle chiamate alla `fopen` combinate con la dichiarazione di `fp`:

```

FILE *fp = fopen(FILE_NAME, "r");
oppure combinate con un controllo con la macro NULL:
if ((fp = fopen(FILE_NAME, "r")) == NULL) ...

```

Collegare un file con uno stream aperto

```

FILE *freopen(const char *restrict filename,
              const char *restrict mode,
              FILE * restrict stream);

```

freopen La funzione `freopen` collega un file diverso a uno stream che è già stato aperto. L'uso più comune di questa funzione è quella di associare un file con uno degli stream standard (`stdin`, `stdout` o `stderr`). Per esempio, per fare in modo che un programma inizi a scrivere sul file `foo`, possiamo utilizzare la seguente chiamata alla funzione `freopen`.

```

if (freopen("foo", "w", stdout) == NULL) {
    /* errore; foo non può essere aperto */
}

```

Dopo aver chiuso qualsiasi file precedentemente associato a `stdout` (per mezzo del reindirizzamento dell'output o attraverso una chiamata precedente alla `freopen`), la funzione `freopen` aprirà il file `foo` e lo assocerà allo stream `stdout`.

Il valore normalmente restituito dalla `freopen` consiste del suo terzo argomento (un puntatore a file). Se non può aprire il nuovo file, la funzione restituirà un puntatore nullo (la `freopen` ignora l'errore se il vecchio file non può essere chiuso).

 Il C99 aggiunge un nuovo colpo di scena. Se l'argomento `filename` è un puntatore nullo, allora la `freopen` cerca di modificare la modalità di apertura dello stream facendola diventare come quella specificata dal parametro `mode`. Tuttavia alle diverse implementazioni non viene richiesto di supportare questa nuova funzionalità e se lo facessero potrebbero imporre delle restrizioni su quali modifiche possano essere apportate alla modalità di apertura.

Ottenere i nomi dei file dalla riga di comando

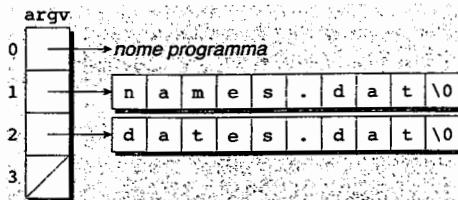
Quando scriviamo un programma che deve aprire un file, si presenta immediatamente un problema: come forniamo il nome del file al programma? Costruire i nomi dei file all'interno del programma non conferisce molta flessibilità e chiedere all'utente di immettere i nomi dei file può essere scomodo. Spesso la soluzione migliore è quella di fare in modo che i programmi ottengano i nomi dei file a partire dalla riga di comando. Per esempio: quando eseguiamo un programma chiamato demo, possiamo fornirgli i nomi dei file inserendoli nella riga di comando:

```
demo names.dat dates.dat
```

Nella Sezione 13.7 abbiamo visto come accedere agli argomenti della riga di comando definendo il `main` come una funzione con due parametri:

```
int main(int argc, char *argv[])
{
}
```

Il parametro `argc` corrisponde al numero di argomenti della riga di comando, mentre `argv` è un vettore di puntatori alle stringhe contenenti tali argomenti. L'elemento `argv[0]` punta al nome del programma, gli elementi da `argv[1]` ad `argv[argc-1]` puntano ai restanti argomenti. L'elemento `argv[argc]` è un puntatore nullo. Nell'esempio precedente, `argc` è pari a 3, `argv[0]` punta a una stringa contenente il nome del programma, `argv[1]` punta alla stringa "names.dat" e `argv[2]` punta alla stringa "dates.dat":



Controllare se un file può essere aperto

Il programma seguente determina se un file esiste e se può essere aperto in lettura. Quando il programma viene eseguito, l'utente gli passa il nome del file da controllare:

```
canopen file
```

Il programma stamperà il messaggio "file can be opened" oppure il messaggio "file can't be opened". Se l'utente immette sulla riga di comando un numero non corretto di argomenti, il programma stamperà il messaggio "usage: canopen filename" per ricordare all'utente che canopen necessita di un unico nome file.

```
canopen.c /* Controlla se un file può essere aperto in lettura */

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

int main(int argc, char *argv[])
{
    FILE *fp;

    if (argc != 2) {
        printf("usage: canopen filename\n");
        exit(EXIT_FAILURE);
    }

    if ((fp = fopen(argv[1], "r")) == NULL) {
        printf("%s can't be opened\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    printf("%s can be opened\n", argv[1]);
    fclose(fp);
    return 0;
}
```

Osservate che possiamo utilizzare il reindirizzamento per scartare l'output di canopen e controllare semplicemente il valore di stato che restituisce.

File temporanei

```
FILE *tmpfile(void);
char *tmpnam(char *s);
```

Spesso i programmi del mondo reale hanno bisogno di creare dei file temporanei (file che esistono solo fintanto che il programma è in esecuzione). I compilatori C, per esempio, creano spesso dei file temporanei. Un compilatore può inizialmente tradurre un programma C in qualche forma intermedia che salverà all'interno di un file. Successivamente il compilatore potrà rileggere il contenuto di quel file nel momento in cui traduce il programma in codice oggetto. Una volta che il programma è stato compilato completamente, non c'è bisogno di conservare il file che contiene la forma intermedia. L'header `<stdio.h>` fornisce due funzioni, `tmpfile` e `tmpnam`, per la gestione dei file temporanei.

`tmpfile` La funzione `tmpfile` crea un file temporaneo (aperto in modalità "`wb+`") che esisterà fino a quando non verrà chiuso o il programma avrà termine. Una chiamata a questa funzione restituisce un puntatore a file che può essere successivamente utilizzato per accedere al file:

```
FILE *tempptr;
tempptr = tmpfile(); /* crea un file temporaneo */
```

Se la creazione del file non ha buon esito, la funzione `tmpfile` restituisce un puntatore nullo.

Sebbene la `tmpfile` sia facile da usare, presenta un paio di inconvenienti: (1) non possiamo conoscere il nome del file che viene creato dalla funzione, e (2) non possiamo decidere in un secondo momento di rendere il file permanente. Se queste due restrizioni si rivelassero un problema, l'alternativa sarebbe quella di creare il file temporaneo usando la funzione `fopen`. Naturalmente non vogliamo che questo abbia lo stesso nome di un file precedentemente esistente, di conseguenza abbiamo bisogno di un modo per generare dei nuovi nomi per i file e qui entra in gioco la funzione `tmpnam`.

tmpnam La funzione `tmpnam` genera un nome per un file temporaneo. Se il suo argomento è un puntatore nullo, `tmpnam` salva il nome del file in una variabile statica e restituisce un puntatore a quest'ultima:

```
char *filename;
-
filename = tmpnam(NULL); /* crea il nome per un file temporaneo */
```

Altrimenti la funzione copia il nome del file all'interno del vettore di caratteri indicato dal programmatore:

```
char filename[L_tmpnam];
-
tmpnam(filename); /* crea il nome per un file temporaneo */
```

Nell'ultimo caso, la funzione restituisce anche un puntatore al primo carattere di questo vettore. `L_tmpnam` è una macro presente in `<stdio.h>` che specifica quanto debba essere lungo un vettore di caratteri che deve contenere il nome di un file temporaneo.



Assicuratevi che l'argomento della funzione `tmpnam` punti a un vettore di almeno `L_tmpnam` caratteri. Fate attenzione anche a non chiamare troppo spesso la funzione `tmpnam`. La macro `TMP_MAX` (definita in `<stdio.h>`) specifica il numero massimo di nomi di file temporanei che possono essere potenzialmente generati dalla `tmpnam` durante l'esecuzione del programma. Se non è in grado di generare un nome file, la `tmpnam` restituisce un puntatore nullo.

File buffering

```
int fflush(FILE *stream),
void setbuf(FILE * restrict stream,
            char * restrict buf);
int setvbuf(FILE * restrict stream,
            char * restrict buf,
            int mode, size_t size);
```

Trasferire dati da un disco o in un disco è un'operazione relativamente lenta. Di conseguenza, non è pensabile che un programma possa accedere a un file su disco ogni volta che volesse leggere o scrivere un byte. Il segreto per raggiungere delle

performance accettabili è il **buffering**: i dati scritti su uno stream vengono di fatto mantenuti in un'area nella memoria. Quando quest'area è piena (o lo stream viene chiuso), avviene il cosiddetto **flush** del buffer (il buffer viene svuotato scrivendo il suo contenuto nel dispositivo di output). Gli stream di input possono essere bufferizzati in modo simile: il buffer contiene i dati provenienti dal dispositivo di input e l'input viene letto dal buffer invece che dal dispositivo vero e proprio. La tecnica del buffering può comportare un enorme guadagno dal punto di vista dell'efficienza, dato che leggere un byte da un buffer o scrivere un byte al suo interno è praticamente istantaneo. Naturalmente ci vuole del tempo per trasferire il contenuto del buffer sul disco, ma effettuare un unico spostamento "in blocco" è molto più veloce di eseguire tanti piccoli trasferimenti della dimensione di un byte.

Le funzioni presenti in <stdio.h> eseguono automaticamente il buffering quando questo sembra vantaggioso. Il buffering avviene dietro le quinte e di solito non dobbiamo preoccuparcene. In certe rare occasioni tuttavia, potremmo aver bisogno di intraprendere un ruolo più attivo. Se così fosse, potremmo usare le funzioni ffflush, setbuf e setvbuf.

fflush Quando un programma scrive dell'output su un file, di solito i dati vanno inizialmente a finire all'interno di un buffer. Il buffer viene svuotato automaticamente quando è pieno o quando il file viene chiuso. Chiamando la funzione fflush tuttavia, un programma può svuotare un buffer ogni volta che vuole. La chiamata

```
fflush(fp); /* svuota il buffer del file fp */
```

svuota il buffer associato al file fp. La chiamata

```
fflush(NULL); /* svuota tutti i buffer */
```

svuota tutti gli stream di output. La funzione fflush restituisce uno zero se va a buon fine, mentre restituisce EOF se si verifica qualche errore.

setvbuf La funzione setvbuf ci permette di modificare il modo in cui uno stream viene bufferizzato e di controllare la dimensione e la posizione del buffer. Il terzo argomento della funzione specifica il tipo di buffering desiderato, il quale deve corrispondere una delle seguenti macro:

- **_IOFBF (full buffering)**. I dati vengono letti dallo stream quando il buffer è vuoto e vengono scritti nello stream quando è pieno.
- **_IOLBF (line buffering)**. I dati vengono letti dallo stream o scritti nello stream una riga alla volta.
- **_IONBF (no buffering)**. I dati vengono letti dallo stream o scritti sullo stream direttamente, senza l'uso di un buffer.

Tutte e tre le macro vengono definite all'interno di <stdio.h>. La modalità di **full buffering** è quella di default per gli stream che non sono connessi a dispositivi interattivi.

Il secondo argomento della setvbuf (nel caso non fosse un puntatore nullo) rappresenta l'indirizzo del buffer desiderato. Il buffer può avere una durata di memorizzazione statica, dinamica o persino essere allocata dinamicamente. Rendere un buffer automatico fa sì che il suo spazio venga reclamato automaticamente all'uscita dal blocco. Allocare dinamicamente lo spazio per il buffer ci permette di rilasciare il buffer quando questo non fosse più necessario. L'ultimo argomento della funzione



`setvbuf` rappresenta il numero di byte presenti nel buffer. Un buffer più grande può garantire delle performance migliori, un buffer più piccolo può farci risparmiare spazio.

La seguente chiamata alla `setvbuf`, per esempio, modifica il buffering di `stream` nella modalità full buffering utilizzando come buffer gli `N` byte del vettore `buffer`:

```
char buffer[N];
-
setvbuf(stream, buffer, _IOFBF, N);
```

 La funzione `setvbuf` deve essere chiamata dopo che lo stream è stato aperto, ma prima delle altre operazioni eseguite su quest'ultimo.

È possibile anche chiamare la `setvbuf` usando un puntatore nullo come secondo argomento, il che richiede che la funzione crei un buffer della dimensione specificata. La funzione restituisce uno zero nel caso abbia successo. Restituisce un valore diverso da zero se l'argomento `mode` non è valido o se la richiesta non può essere assecondata.

`setbuf` è una funzione più vecchia che assume i valori di default per la modalità di buffering e la dimensione del buffer. Se l'argomento `buf` è un puntatore nullo, la chiamata `setbuf(stream, buf)` è equivalente a

```
(void) setvbuf(stream, NULL, _IONBF, 0);
oppure è equivalente a
(void) setvbuf(stream, buf, _IOFBF, BUFSIZ);
```

dove `BUFSIZ` è una macro definita in `<stdio.h>`. La funzione `setbuf` è considerata obsoleta e il suo utilizzo non è raccomandato nei nuovi programmi.

 Quando utilizzate la `setvbuf` e la `setbuf`, assicuratevi di chiudere lo stream prima che il suo buffer venga deallocated. In particolare, se il buffer è locale per una funzione e possiede una durata di memorizzazione automatica, assicuratevi di chiudere lo stream prima che la funzione abbia termine.

Operazioni varie sui file

```
int remove (const char *filename);
int rename (const char *old, const char *new);
```

Le funzioni `remove` e `rename` permettono a un programma di eseguire operazioni basili di gestione dei file. A differenza della maggior parte delle altre funzioni di questa sezione, `remove` e `rename` lavorano con i *nomi* dei file e non con dei *puntatori*. Entrambe le funzioni restituiscono uno zero nel caso le loro chiamate abbiano successo, mentre

restituiscono un valore diverso da zero nel caso le chiamate non andassero a buon fine.

remove La funzione `remove` cancella un file:

```
remove("foo"); /* cancella il file chiamato "foo" */
```

Se un programma utilizza la `fopen` (invece della `tmpfile`) per creare un file temporaneo, può utilizzare la funzione `remove` per cancellare il file prima che termini il programma. Assicuratevi che il file che deve essere rimosso sia stato chiuso. L'effetto dell'eliminazione di un file correntemente aperto è definito dall'implementazione.

rename La funzione `rename` modifica il nome di un file:

```
rename("foo", "bar"); /* rinomina "foo" in "bar" */
```

la funzione `rename` è comoda per rinominare un file temporaneo creato utilizzando la `fopen`, nel caso in cui il programma decidesse di renderlo permanente. Se esiste già un file con il nuovo nome, allora l'effetto che si ottiene è definito dall'implementazione.



Se il file che deve essere rinominato è aperto, assicuratevi di chiuderlo prima di chiamare `rename`. La chiamata della funzione potrebbe non andare a buon fine nel caso le si chiedesse di rinominare un file aperto.

22.3 I/O formattato

In questa sezione esamineremo le funzioni di libreria che utilizzano delle stringhe di formato per controllare la lettura e la scrittura. Queste funzioni, che includono le nostre vecchie conoscenze `printf` e `scanf`, possiedono l'abilità di convertire i dati dalla forma testuale a quella numerica durante l'input, e dalla forma numerica a quella testuale durante le operazioni di output. Nessuna delle altre funzioni di I/O è in grado di effettuare questo tipo di conversioni.

Le funzioni ...printf

```
int fprintf(FILE * restrict stream,
            const char * restrict format, ...);
int printf(const char * restrict format, ...);
```

fprintf
printf

Le funzioni `fprintf` e `printf` scrivono un numero variabile di dati nello stream di output utilizzando una stringa di formato che controlla il modo di presentarsi dell'output. I prototipi per entrambe le funzioni terminano con il simbolo ... (un'ellisse [ellisse > 26.1]). Questo simbolo indica che la funzione possiede un numero variabile di argomenti aggiuntivi. Entrambe le funzioni restituiscono il numero di caratteri scritti. Un valore restituito negativo indica che si è verificato un errore.

L'unica differenza tra la `printf` e la `fprintf` è che la prima scrive sempre su `stdout` (lo stream di standard output), mentre la `fprintf` scrive sullo stream indicato dal suo primo argomento:

```
printf("Total: %d\n", total); /* scrive su stdout */
fprintf(fp, "Total: %d\n", total); /* scrive su fp */
```

Una chiamata alla printf è equivalente a una chiamata alla fprintf con stdout come primo argomento.

Non pensate alla fprintf semplicemente come a una funzione che scrive dei dati su file. Come molte altre delle funzioni presenti nell'header <stdio.h>, anche questa funziona perfettamente con qualsiasi tipo di stream di output. Infatti, uno degli utilizzi più comuni della funzione fprintf (scrivere un messaggio di errore su stderr, lo stream standard per gli errori) non ha nulla a che fare con i file su disco. Ecco come potrebbe presentarsi una chiamata di questo tipo:

```
fprintf(stderr, "Error: data file can't be opened.\n");
```

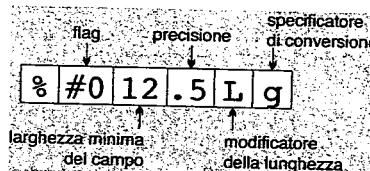
Scrivere il messaggio su stderr garantisce che questo comparirà sullo schermo anche nel caso in cui l'utente reindirizzasse lo standard output.

Nell'header <stdio.h> sono presenti altre due funzioni che possono scrivere dell'output formattato su uno stream. Queste funzioni chiamate vfprintf e vprintf sono piuttosto oscure [[funzioni v...printf > 26.1](#)], entrambe si basano sul tipo va_list, che viene dichiarato all'interno dell'header <stdarg.h>; per questo motivo le discuteremo unitamente a quell'header.

Specifiche di conversione per le funzioni ...printf

Sia la printf che la fprintf richiedono una stringa di formato contenente dei normali caratteri e/o delle specifiche di conversione. I normali caratteri vengono stampati così come sono. Le specifiche di conversione descrivono come i restanti argomenti debbano essere convertiti alla forma testuale. Le specifiche di conversione sono state descritte brevemente nella Sezione 3.1, inoltre maggiori dettagli sono stati aggiunti nei capitoli successivi. Ora rivedremo quanto già conosciamo sulle specifiche di conversione e riempiremo le lacune rimaste.

Una specifica di conversione per una funzione ...printf è costituita dal simbolo % seguito da fino a cinque oggetti distinti:



Ecco una descrizione dettagliata di questi oggetti che devono comparire nell'ordine illustrato.

- **Flag** (opzionale, ne è permesso più di uno). Il flag - fa sì che all'interno del campo venga adottato l'allineamento a sinistra. Gli altri flag hanno a che fare con il modo in cui i numeri vengono visualizzati. La Tabella 22.4 fornisce la lista completa dei flag.

Tabella 22.4 Flag per le funzioni ...printf

Flag	Significato
-	Allinea a sinistra all'interno del campo (per default viene applicato l'allineamento a destra).
+	I numeri prodotti dalle conversioni con segno iniziano sempre con un + o con un - (normalmente solo i numeri negativi vengono preceduti dal segno).
spazio	I numeri non negativi prodotti dalle conversioni con segno vengono preceduti da uno spazio (il flag + annulla il flag spazio).
#	I numeri ottali iniziano con lo 0, i numeri esadecimali diversi da zero iniziano con 0x o ox. I numeri a virgola mobile hanno sempre il separatore decimale. Gli zeri trascinati non vengono sempre rimossi dai numeri stampati con le conversioni g e G.
0 (zero)	I numeri vengono gonfiati (<i>padding</i>) con degli zeri fino a riempire tutta la larghezza del campo. Il flag 0 viene ignorato se lo specificatore di conversione è uno tra d, i, o, u, x o X, e se è stata specificata la precisione (il flag - annulla l'effetto del flag 0).

- **Larghezza minima del campo** (opzionale). Un oggetto che fosse troppo piccolo per occupare questo numero di caratteri verrebbe "gonfiato" (per default vengono aggiunti degli spazi alla sinistra dell'oggetto, allineandolo così a destra all'interno del campo). Un oggetto che fosse troppo grande per la larghezza del campo verrebbe visualizzato comunque nella sua interezza. La larghezza del campo può essere sia un intero che il carattere *. Se il carattere * è presente, la larghezza del campo viene ottenuta dall'argomento successivo. Se questo argomento è negativo, viene trattato come un numero positivo preceduto dal flag -.

- **Precisione** (opzionale). Il significato della precisione dipende dalla conversione usata:

d, i, o, u, x, X: numero minimo di cifre (degli zeri vengono aggiunti nel caso in cui il numero avesse meno cifre)

a, A, e, E, f, F: numero di cifre dopo il separatore decimale

g, G: numero di cifre significative

s: numero massimo di byte

La precisione è costituita da un punto (.) seguito da un intero o dal carattere *. Se è presente il carattere *, la precisione viene ottenuta dall'argomento successivo (se questo argomento è negativo, l'effetto è lo stesso di non aver specificato la precisione). Se è presente solo il punto la precisione è pari a zero.

Modificatore della lunghezza (opzionale). La presenza di un modificatore della lunghezza indica che l'oggetto che deve essere visualizzato è di un tipo che è più lungo o più corto del normale rispetto alla particolare specifica di conversione in uso (per esempio, normalmente %d si riferisce a un valore int, %hd viene utilizzato per visualizzare uno short int e %ld viene usato per i long int). La Tabella 22.5 elenca tutti

i modificatori di lunghezza, gli specificatori di conversione con i quali questi possono essere usati e il tipo indicato dalla loro combinazione (una qualsiasi combinazione di modificatore di lunghezza e specificatore di conversione non indicata in tabella provoca un comportamento indefinito).

Tabella 22.5 Modificatori di lunghezza per le funzioni ...printf

Modificatore di lunghezza	Specificatore di conversione	Significato
hh ^t	d, i, o, u, x, X	signed char, unsigned char
	n	signed char *
h	d, i, o, u, x, X	short int, unsigned short int
	n	short int *
l (ell)	d, i, o, u, x, X	long int, unsigned long int
	n	long int *
	c	wint_t
	s	wchar_t *
a, A, e, E, f, F, g, G		nessun effetto
ll ^t (ell-ell)	d, i, o, u, x, X	long long int, unsigned long long int
	n	long long int *
j ^t	d, i, o, u, x, X	intmax_t, uintmax_t
	n	intmax_t *
z ^t	d, i, o, u, x, X	size_t
	n	size_t *
t ^t	d, i, o, u, x, X	ptrdiff_t
	n	ptrdiff_t *
L	a, A, e, E, f, F, g, G	long double

^tsolo C99

- Specificatore di conversione.** Lo specificatore di conversione deve essere uno dei caratteri contenuti nella Tabella 22.6. Osservate che f, F, e, E, g, G, a e A sono tutti pensati per scrivere valori di tipo double, tuttavia funzionano anche per i valori float. Grazie alle promozioni di default degli argomenti [promozioni di default degli argomenti > 9.3], gli argomenti float vengono automaticamente convertiti al tipo double quando vengono passati a una funzione che ha un numero variabile di argomenti. Analogamente un carattere passato alla ...printf viene convertito automaticamente al tipo int e quindi la conversione c funziona a dovere.

Fate attenzione a seguire scrupolosamente le regole appena descritte. L'utilizzo di una specifica di conversione non valida causa un comportamento indefinito.



Tabella 22.6 Specificatori di conversione per le funzioni ...printf

Specificatore di conversione	Significato
d, i	Converte un valore int nel formato decimale
o, u, x, X	Converte un valore unsigned int in base 8 (o), base 10 (u) 16 (x, X). Lo specificatore x visualizza le cifre esadecimali a-f lettere minuscole, mentre X le visualizza come lettere maiuscole.
f, F ^t	Converte un valore double nella forma decimale mettendo il separatore decimale nella posizione corretta. Se non specificata nessuna precisione, visualizza sei cifre dopo il segnatore decimale.
e, E	Converte un valore double nella notazione scientifica. Se viene specificata nessuna precisione, visualizza sei cifre dopo il separatore decimale. Se viene scelto e, l'esponente viene prodotto dalla lettera e. Se viene scelto E, l'esponente viene prodotto dalla lettera E.
g, G	Lo specificatore g converte un valore double o nel formato f o nel formato e. Il formato e viene scelto quando l'esponente è maggiore di -4 oppure è maggiore o uguale alla precisione. Gli zeri seguenti non vengono visualizzati (a meno che non venga usato il flag). Il separatore decimale è presente solo quando è seguito da una virgola. Lo specificatore G sceglie tra i formati F ed E.
a ^t , A ^t	Converte un valore double nella notazione scientifica esamale utilizzando la forma [-]oxh.hhhptd, dove [-] è un segno opzionale, le h rappresentano delle cifre esadecimali, t è un segno più oppure quello meno e d è l'esponente. d è un numero decimale che rappresenta una potenza di 2. Se la precisione non viene specificata, dopo il punto viene visualizzata una numero sufficiente di cifre per rappresentare il valore esattamente (se possibile). Lo specificatore a visualizza le cifre decimali a-f come lettere minuscole, mentre A come delle lettere maiuscole. La scelta di a o A ha effetto anche sulle lettere x e X.
c	Visualizza un valore int come un carattere senza segno.
s	Scrive i caratteri puntati dall'argomento. Interrompe la scrittura quando viene raggiunto il numero di byte specificati dalla dimensione (se presente) o quando viene incontrato il carattere \0.
p	Converte un valore void * in forma stampabile.
n	L'argomento corrispondente deve puntare a un oggetto di tipo int. Lo specificatore fa salvare in questo oggetto il numero di caratteri scritti fino a quel momento dalla chiamata alla ...printf.
%	Non produce output.
%%	Scrive il carattere %.

C99

Modifiche del C99 alle specifiche di conversione ...printf

Nello standard C99 le specifiche di conversione della printf e della fprintf hanno subito un certo numero di modifiche:

- **Modificatori di lunghezza aggiuntivi.** Il C99 introduce i modificatori lunghezza hh, ll, j, z e t. I modificatori hh e ll forniscono delle opzioni aggiuntive sulla lunghezza, j permette di scrivere gli interi della dimensione più grande [interi della dimensione più grande > 27.1], mentre z e t facilitano rispettivamente la scrittura dei tipi size_t e ptrdiff_t.
- **Specificatori di conversione aggiuntivi.** Il C99 introduce gli specificatori a e A. F è equivalente a f ad eccezione del modo in cui vengono scritti i valori infinito e NaN (leggente più avanti). Le specifiche di conversione a e A vengono usate raramente. Sono relative alle costanti esadecimali a virgola mobile che vengono trattate nella Sezione Domande & Risposte del Capitolo 7.
- **Possibilità di scrivere infinito e NaN.** Lo standard floating point IEEE 754 [standard floating point IEEE 754 > 23.4] ammette che il risultato di un'operazione a virgola mobile sia infinito (*infinity*), infinito negativo (*negative infinity*) o NaN ("not a number"). Per esempio, dividendo 1.0 per 0.0 otteniamo un infinito positivo, dividendo -1.0 per 0.0 otteniamo infinito negativo e dividendo 0.0 per 0.0 otteniamo NaN (perché il risultato non è matematicamente indefinito). Nel C99 gli specificatori di conversione a, A, e, E, f, F, g e G sono in grado di convertire questi speciali valori in una forma che può essere visualizzata. Gli specificatori e, f e g convertono l'infinito positivo in inf o infinity (sono ammessi entrambi) l'infinito negativo in -inf o -infinity e NaN in nan o -nan (eventualmente seguiti da una serie di caratteri racchiusi tra parentesi). Gli specificatori A, E, F e G sono equivalenti ai precedenti ma utilizzano le lettere maiuscole (INF, INFINITY, NAN).
- **Supporto per i wide character.** Un'altra caratteristica del C99 è data dalla capacità della funzione fprintf di scrivere i wide character [wide character > 25.2]. La specifica di conversione %lc viene utilizzata per scrivere un singolo wide character, mentre la specifica %ls viene utilizzata per le stringhe di wide character.
- **Ora sono ammesse le specifiche non definite precedentemente.** Nel C89 l'effetto provocato dall'utilizzo delle specifiche %le, %lf, %lg e %lg è indefinito. Queste specifiche sono ammesse invece nel C99 (il modificatore di lunghezza viene semplicemente ignorato).

Esempi per le specifiche di conversione delle funzioni ...printf

Finalmente è il momento di fare alcuni esempi. Nei capitoli precedenti abbiamo visto parecchie specifiche di conversione per gli usi più comuni, di conseguenza oggi ci concentreremo su quelle più avanzate. Come nei capitoli precedenti utilizzeremo il carattere • per rappresentare il carattere spazio.

Iniziamo esaminando l'effetto dei flag sulla conversione %d (sulle altre conversioni hanno un effetto simile). La prima riga della Tabella 22.7 mostra l'effetto della conversione %8d senza alcun flag. Le quattro righe successive illustrano l'effetto dei flag -, +, spazio e 0 (il flag # non viene mai utilizzato con %d). Le restanti righe illustrano l'effetto derivato dalla combinazione dei flag.

Tabella 22.7 Effetto dei flag sulla conversione %d

Specifiche di conversione	Risultato applicando la conversione a 123	Risultato applicando la conversione a -123
%8d	*****123	*****-123
%-8d	123*****	-123*****
%+8d	*****+123	*****-123
% 8d	*****123	*****-123
%08d	00000123	-00000123
%-+8d	+123*****	-123*****
%- 8d	*123*****	-123*****
%+08d	12300000	-12300000
% 08d	*0000123	-12300000

La Tabella 22.8 illustra l'effetto del flag # sulle conversioni o, x, X, g e G.

Tabella 22.8 Effetto del flag #

Specifiche di conversione	Risultato applicando la conversione a 123	Risultato applicando la conversione a 123.0
%8o	*****173	*****173
%#8o	****0173	****0173
%8x	*****7b	*****7b
%#8x	****0x7b	****0x7b
%8X	*****7B	*****7B
%#8X	****0X7B	****0X7B
%8g		*****123
%#8g		*123.000
%8G		*****123
%#8G		*123.000

Nei capitoli precedenti per la stampa dei numeri abbiamo utilizzato sia la larghezza minima del campo che la precisione e quindi non c'è motivo di fornire degli esempi qui. La Tabella 22.9 invece, illustra l'effetto sulla conversione %s della larghezza minima del campo e la precisione.

Tabella 22.9 Effetto della larghezza minima del campo e della precisione sulla conversione %s

Specifiche di conversione	Risultato applicando la conversione a "bogus"	Risultato applicando la conversione a "buzzword"
%6s	•bogus	buzzword
%-6s	bogus•	buzzword
.4s	bogu	buzz
%6.4s	••bogu	••buzz
%-6.4s	bogu••	buzz••

La Tabella 22.10 illustra come la conversione %g visualizzi alcuni numeri nel formato %e e altri nel formato %f. Tutti i numeri presenti nella tabella vengono scritti usando la specifica di conversione %.4g. I primi due numeri hanno un esponente almeno pari a 4 e quindi verranno visualizzati nel formato %e. Gli otto numeri successivi vengono visualizzati nel formato %f. Gli ultimi due numeri hanno un esponente inferiore a -4, di conseguenza vengono visualizzati nel formato %e.

Tabella 22.10 Esempi della conversione %g

Numero	Risultato applicando la conversione %.4g al numero
123456.	1.235e+05
12345.6	1.235e+04
1234.56	1235
123.456	123.5
12.3456	12.35
1.23456	1.235
.123456	0.1235
.0123456	0.01235
.00123456	0.001235
.000123456	0.0001235
.0000123456	1.235e-05
.00000123456	1.235e-06

In passato abbiamo assunto che la larghezza minima del campo e la precisione fossero delle costanti incorporate all'interno della stringa di formato. Mettendo il carattere * al posto di uno di questi due numeri normalmente ci permette di specificarli attraverso un argomento posto *dopo* la stringa di formato. Le seguenti chiamate alla printf, per esempio, producono tutte lo stesso output:

```
printf("%6.4d", i);
printf("%*.4d", 6, i);
printf("%6.*d", 4, i);
printf("%*.*d", 6, 4, i);
```

Osservate come i valori che devono sostituire il carattere * si trovino immediatamente prima del valore che deve essere visualizzato. In ogni caso uno dei maggiori van-

taggi nell'uso del carattere * è che ci permette di utilizzare una macro per specificare la larghezza della precisione:

```
printf("%*d", WIDTH, i);
```

Possiamo persino calcolare la larghezza o la precisione durante l'esecuzione del programma:

```
printf("%*d", page_width / num_cols, i);
```

Le specifiche meno comuni sono la %p e la %n. La conversione %p ci permette di stampare il valore di un puntatore:

```
printf("%p", (void *) ptr); /* visualizza il valore di ptr */
```

Sebbene la specifica %p sia occasionalmente utile durante il debugging, non è una delle caratteristiche più utilizzate dai programmati nelle loro mansioni quotidiane. Lo standard C non specifica come un puntatore debba presentarsi quando viene stampato per mezzo della specifica %p, tuttavia è probabile che questo venga visualizzato come un numero ottale o esadecimale.

La conversione %n viene utilizzata per scoprire il numero di caratteri stampati fino a quel momento dalla chiamata alla ...printf. Per esempio, dopo la chiamata

```
printf("%d%n\n", 123, &len);
```

il valore di len diventerà uguale a 3, visto che printf avrà scritto 3 caratteri (123) nel momento in cui ha ricevuto la specifica %n. Osservate che la variabile len deve essere preceduta dal simbolo & (perché %n richiede un puntatore) e che la stessa len non viene stampata.

Le funzioni ...scanf

```
int fscanf(FILE * restrict stream,
           const char * restrict format, ...);
int scanf(const char * restrict format, ...);
```

Le funzioni fscanf e scanf leggono dei dati dallo stream di input utilizzando una stringa di formato per indicare la disposizione dell'input stesso. Dopo la stringa di formato segue un numero qualsiasi di puntatori (ognuno che punta a un oggetto). Gli oggetti di input vengono convertiti (in accordo con le specifiche di conversione presenti nella stringa di formato) e salvati in questi oggetti.

La scanf legge sempre da stdin (lo stream di input standard), mentre la fscanf legge dallo stream indicato dal suo primo argomento:

```
scanf("%d%d", &i, &j);          /* legge da stdin */
fscanf(fp, "%d%d", &i, &j);      /* legge da fp */
```

Una chiamata alla scanf è equivalente a una chiamata alla fscanf con stdin come primo argomento.

Le funzioni ...scanf terminano prematuramente se si verifica un **input failure** (non possono essere letti altri caratteri di input) o un **matching failure** (i caratteri in input non si adattano alla stringa di formato). Nel C99 un input failure può verificarsi anche a causa di un **errore di codifica**, ovvero se c'è stato un tentativo di leggere un

carattere multibyte [**caratteri multibyte > 25.2**] ma i caratteri in input non corrispondono ad alcun carattere multibyte valido. Entrambe le funzioni restituiscono il numero di dati che sono stati letti e assegnati agli oggetti. Le funzioni restituiscono EOF nel caso si verificasse un input failure prima che uno qualsiasi dei dati possa essere letto.

I cicli che controllano il valore restituito dalla scanf sono molto comuni nei programmi C. Il ciclo seguente, per esempio, legge una serie di numeri interi uno alla volta, fermandosi al primo segno di problemi:

```
while (scanf("%d", &i) == 1) {  
}
```

Stringhe di formato per le funzioni ...scanf

Le chiamate alle funzioni ...scanf somigliano a quelle delle funzioni ...printf. Questa somiglianza può essere fuorviante, le funzioni ...scanf agiscono in modo piuttosto diverso dalle ...printf. La cosa migliore è pensare alla scanf e alla fscanf come a delle funzioni di *pattern-matching*. La stringa di formato rappresenta un *pattern* che la funzione ...scanf cerca di far combaciare con l'input che legge. Se l'input non si adatta alla stringa di formato, allora la funzione termina non appena se ne accorge. Il carattere di input che non combacia viene "rimesso a posto" in modo che possa essere letto in futuro.

Una stringa di formato per la ...scanf contiene tre cose:

- **Specifiche di conversione.** Le specifiche di conversione presenti nella stringa di formato della ...scanf ricordano quelle delle stringhe di formato della ...printf. La maggior parte delle specifiche salta i caratteri di spazio bianco [**caratteri di spazio bianco > 3.2**] presenti all'inizio di un oggetto di input (le eccezioni sono %[, %c e %n). Le specifiche di conversione non saltano mai i caratteri di spazio bianco che seguono l'oggetto. Se l'input contiene «123», la specifica di conversione %d consuma i caratteri •, 1, 2 e 3 ma non legge il carattere « (stiamo usando • per rappresentare il carattere di spazio e « per rappresentare il carattere new-line).
- **Caratteri di spazio bianco.** Uno o più caratteri di spazio bianco consecutivi presenti all'interno di una stringa di formato di una ...scanf si accoppiano con zero o più caratteri di spazio bianco presenti nello stream di input.
- **Caratteri non di spazio bianco.** Un carattere che non sia di spazio bianco (eccetto %) si accoppia nello stream di input con lo stesso carattere.

Per esempio, la stringa di formato "ISBN %d-%d-%ld-%d" specifica che l'input considerà di:

le lettere ISBN

qualche possibile carattere di spazio bianco

un intero

il carattere -

un intero (eventualmente preceduto da un carattere di spazio bianco)

il carattere -

un intero di tipo long (eventualmente preceduto da un carattere di spazio bianco)

il carattere -

un intero (eventualmente preceduto da un carattere di spazio bianco)

Specifiche di conversione per la ...scanf

Le specifiche di conversione per le funzioni ...scanf sono effettivamente più semplici rispetto a quelle per le funzioni ...printf. Una specifica di conversione per la ...scanf consiste nel carattere % seguito dagli elementi elencati di seguito (nell'ordine nel quale vengono mostrate).

- * (opzionale). La presenza di * indica una **soppressione dell'assegnamento** (*assignment suppression*): un oggetto di input viene letto ma non assegnato a un oggetto. Gli oggetti consumati usando * non vengono inclusi nel conto restituito dalla ...scanf.
- **Larghezza massima del campo** (opzionale). La larghezza massima del campo pone un limite al numero di caratteri presenti in un oggetto di input. La conversione dell'oggetto termina se questo numero viene raggiunto. I caratteri di spazio bianco saltati all'inizio di una conversione non contano.
- **Modificatore della lunghezza** (opzionale). La presenza di un modificatore della lunghezza indica che l'oggetto nel quale verrà salvato l'input è di un tipo che è più grande o più piccolo del normale per una particolare specifica di conversione. La Tabella 22.11 elenca tutti i modificatori della lunghezza, gli specificatori con i quali possono essere utilizzati e il tipo indicato dalla combinazione dei due (una qualsiasi combinazione tra il modificatore della lunghezza e lo specificatore di conversione non illustrata nella tabella provoca un comportamento indefinito).

Tabella 22.11 Modificatori della lunghezza per le funzioni ...scanf

Modificatore della lunghezza	Specificatore di conversione	Significato
hh†	d, i, o, u, x, X, n	signed char *, unsigned char *
h	d, i, o, u, x, X, n	short int *, unsigned short int *
l (ell)	d, i, o, u, x, X, n	long int *, unsigned long int *
	a, A, e, E, f, F, g, G	double *
	c, s, o [wchar_t *
ll† (ell-ell)	d, i, o, u, x, X, n	long long int *, unsigned long long int *
j†	d, i, o, u, x, X, n	intmax_t *, uintmax_t *
z†	d, i, o, u, x, X, n	size_t *
t†	d, i, o, u, x, X, n	ptrdiff_t *
L	a, A, e, E, f, F, g, G	long double *

†solo C99

- **Specificatore di conversione.** Lo specificatore di conversione deve essere uno dei caratteri elencati in Tabella 22.12.

Tabella 22.12 Specificatori di conversione per le funzioni ...scanf

Specificatore di conversione	Significato
d	Si combina con un intero in base 10. Si assume che l'argomento corrispondente sia di tipo int *.
i	Si combina con un intero. Assume che l'argomento corrispondente sia di tipo int *. Si assume anche che l'intero sia in base 10 a meno che non inizi con uno zero (indicando la base ottale) o con 0x o 0X (esadecimale).
o	Si combina con un intero ottale. Assume che l'argomento corrispondente sia di tipo unsigned int *.
u	Si combina con un intero in base 10. Assume che l'argomento corrispondente sia di tipo unsigned int *.
x, X	Si combina con un intero esadecimale. Assume che l'argomento corrispondente sia di tipo unsigned int *.
a ^t , A ^t , e, E, f, F ^t , g, G	Si combina con un numero in virgola mobile. Si assume che l'argomento corrispondente sia di tipo float *. Nel C99 il numero può essere sia infinito che NaN.
c	Si combina con <i>n</i> caratteri, dove <i>n</i> è la larghezza massima del campo oppure un carattere se la larghezza del campo non è stata specificata. Assume che l'argomento corrispondente sia un puntatore a un vettore di caratteri (o un oggetto carattere nel caso non fosse specificata la larghezza del campo). Non aggiunge il carattere null in fondo.
s	Si combina con una sequenza di caratteri non rappresentanti degli spazi bianchi e aggiunge il carattere null in fondo. Assume che l'argomento corrispondente sia un puntatore a un vettore di caratteri.
[Si combina con una sequenza non vuota di caratteri a partire da uno scanset e poi aggiunge il carattere null in fondo. Assume che l'argomento corrispondente sia un puntatore a un vettore di caratteri.
p	Si combina con valore di un puntatore scritto nella forma che avrebbe usato la ...printf. Assume che l'argomento corrispondente sia un puntatore a un oggetto void*.
n	L'argomento corrispondente deve puntare a un oggetto di tipo int. In questo oggetto salva il numero di caratteri letti dalla chiamata alla ...scanf fino a quel momento. Non viene consumato nessun carattere di input, inoltre non ha effetto sul valore restituito dalla ...scanf.
%	Si combina con il carattere %

^tsolo C99

I dati numerici possono sempre iniziare con il segno (+ o -). Gli specificatori o, u, x e X, tuttavia, convertono il dato nella forma senza segno, per questo di solito non vengono utilizzati per leggere numeri negativi.

Lo specificatore [è una versione più complessa (e più flessibile) dello specificatore s. Una specifica di conversione completa utilizzante [è della forma %[set] oppure %[^set], dove set può essere un qualsiasi insieme di caratteri. Tuttavia se il carattere] appartiene all'insieme di caratteri, questo deve comparire per primo. La specifica %[set] si combina con qualsiasi sequenza di caratteri presenti in set (lo **scanset**). La specifica %[^set] invece, si abbina a una qualsiasi sequenza di caratteri *non* presenti nell'insieme set (in altre parole, lo scanset consiste di tutti i caratteri non presenti in set). Ad esempio, %[abc] si combina con qualsiasi stringa contenente solo le lettere a, b e c, mentre %[^abc] si combina con qualsiasi stringa non contenente i caratteri a, b e c.

Molti degli specificatori di conversione delle funzioni ...scanf sono strettamente collegati alle funzioni di conversione numerica **[funzioni di conversione numerica > 26.2]** presenti in <stdlib.h>. Queste funzioni convertono le stringhe (come "-279") nel loro valore numerico equivalente (-279). Lo specificatore d, per esempio, cerca un segno + o - opzionale, seguito da una serie di cifre decimali. Questo è esattamente lo stesso formato che la funzione strtol richiede quando le viene chiesto di convertire una stringa in un numero decimale. La Tabella 22.13 illustra la corrispondenza tra gli specificatori di conversione e le funzioni di conversione numerica.

Tabella 22.13 Corrispondenza tra gli specificatori di conversione delle funzioni ...scanf e le funzioni di conversione numerica

Specificatore di conversione	Funzione di conversione numerica
d	strtol con 10 come base
i	strtol con 0 come base
o	strtoul con 8 come base
u	strtoul con 10 come base
x, X	strtoul con 16 come base
a, A, e, E, f, F, g, G	strod



Vale sicuramente la pena prestare attenzione durante la scrittura delle chiamate alla scanf. Una specifica di conversione non valida all'interno della stringa di formato per la scanf è pericolosa quanto quella all'interno di una stringa di formato per la printf: entrambe provocano un comportamento indefinito.



Modifiche del C99 alle specifiche di conversione per le funzioni ...scanf

Le specifiche di conversione per le funzioni scanf e fscanf hanno subito delle modifiche nel C99. L'elenco delle modifiche, tuttavia, non è lungo quanto quello per le funzioni ...printf.

- **Modificatori di lunghezza aggiuntivi.** Il C99 introduce i modificatori di lunghezza hh, ll, j, z e t. Questi corrispondono ai modificatori di lunghezza presenti nelle specifiche di conversione per le funzioni ...printf.
- **Specificatori di conversione aggiuntivi.** Il C99 introduce gli specificatori di conversione F, a e A. Questi vengono forniti per simmetria alle funzioni ...printf, tuttavia vengono trattati dalle funzioni ...scanf allo stesso modo degli specificatori e, E, f, g e G.

- **Possibilità di leggere i valori infinito e NaN.** Esattamente come le funzioni ...printf possono scrivere i valori infinito e NaN, le funzioni ...scanf possono leggerli. Per essere letti correttamente, questi valori devono presentarsi nello stesso modo nella quale sarebbero stati scritti dalle funzioni ...printf. Viene ignorato il fatto che le lettere siano maiuscole o minuscole (ad esempio: sia INF che inf verrebbero letti come infinito).
- **Supporto per i wide character.** Le funzioni ...scanf sono in grado di leggere dei caratteri multibyte, i quali vengono poi convertiti in wide character per la memorizzazione. La specifica di conversione %lc viene utilizzata per leggere un singolo carattere multibyte o una sequenza di caratteri multibyte. La specifica %ls viene utilizzata per leggere una stringa di caratteri multibyte (alla fine viene aggiunto il carattere null). Anche le specifiche di conversione %[set] e %[^set] possono leggere una stringa di caratteri multibyte.

Esempi delle specifiche di conversione per le funzioni ...scanf

Le successive tre tabelle contengono degli esempi di chiamate alla funzione scanf. Ogni chiamata viene applicata ai caratteri di input mostrati alla sua destra. I caratteri barrati vengono "consumati" dalla chiamata. A destra dell'input compaiono i valori assunti dalle variabili dopo la chiamata.

Gli esempi presenti in Tabella 22.14 mostrano l'effetto ottenuto combinando specifiche di conversione, caratteri di spazio bianco e caratteri non corrispondenti a spazio bianco. In tre casi alla variabile j non viene assegnato alcun valore, quindi questa mantiene il valore posseduto prima della chiamata alla scanf. Gli esempi della Tabella 22.15 illustrano l'effetto della soppressione dell'assegnamento e la specifica della larghezza del campo. Gli esempi della Tabella 22.16 illustrano gli specificatori di conversione più "esoterici" (i, [ed n).

Tabella 22.14 Esempi di chiamate alla scanf (gruppo 1)

Chiamata alla scanf	Input	Variabili
n = scanf("%d%d", &i, &j);	12*,*34#	n: 1 i: 12 j: non modificato
n = scanf("%d,%d", &i, &j);	12*,*34#	n: 1 i: 12 j: non modificato
n = scanf("%d ,%d", &i, &j);	12*,*34#	n: 2 i: 12 j: 34
n = scanf(" %d, %d", &i, &j);	12*,*34#	n: 1 i: 12 j: non modificato

Tabella 22.15 Esempi di chiamate alla scanf (gruppo 2)

Chiamata alla scanf	Input	Variabili
n = scanf("%*d%d", &i);	12*34#	n: 1 i: 34
n = scanf("%*s%s", str);	My+Fair+Lady#	n: 1 str: "Fair"
n = scanf("%1d%2d%3d", &i, &j, &k);	12345#	n: 3 i: 1 j: 23 k: 45
n = scanf("%2d%2s%2d", &i, str, &j);	123456#	n: 3 i: 12 str: "34" j: 56

Tabella 22.16 Esempi di chiamate alla scanf (gruppo 3)

Chiamata alla scanf	Input	Variabili
n = scanf("%i%1%i", &i, &j, &k);	12+012+0x12#	n: 3 i: 12 j: 10 k: 18
n = scanf("%[0123456789]", str);	123abc#	n: 1 str: "123"
n = scanf("%[0123456789]", str);	abc123#	n: 0 str: non modificato
n = scanf("%[^0123456789]", str);	abc123#	n: 1 str: "abc"
n = scanf("%*d%d%n", &i, &j);	10+20+30#	n: 1 i: 20 j: 5

Rilevare la fine del file e le condizioni di errore

```
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
```

Se chiediamo a una funzione ...scanf di leggere e salvare *n* dati, ci aspettiamo che il suo valore restituito sia uguale a *n*. Se il valore restituito è inferiore a *n*, vuol dire che qualcosa è andato storto. Vi sono tre possibilità.

- **Fine del file.** La funzione ha incontrato la fine del file prima che la stringa di formato fosse stata combinata completamente.

- **Errore in lettura.** La funzione non è stata in grado di leggere i caratteri dallo stream.
- **Errore di incompatibilità.** Un dato di input era nel formato sbagliato. Per esempio, la funzione potrebbe aver incontrato una lettera quando stava cercando il primo numero di un intero.

Come possiamo capire quale tipo di errore si è verificato? In molti casi questo non ha importanza: qualcosa è andato storto e dobbiamo abbandonare il programma. Delle volte, però, avremo bisogno di individuare la ragione del malfunzionamento.

Tutti gli stream possiedono due indicatori: un **indicatore di errore** (*error indicator*) e un **indicatore di end-of-file** (*end-of-file indicator*). Questi indicatori vengono dichiarati nel momento in cui lo stream viene aperto. Non sorprendentemente, incontrare la fine del file causa il set dell'indicatore di end-of-file, mentre un errore in lettura causa il set dell'indicatore di errore (l'indicatore di errore viene impostato anche quando sullo stream si verifica un errore in scrittura). Un errore di incompatibilità non modifica nessuno dei due indicatori.

clearerr

Una volta che l'indicatore di errore o quello di end-of-file è stato impostato, rimane in quello stato fino a quando non viene esplicitamente azzerato, magari con una chiamata alla funzione clearerr. Questa funzione azzera entrambi gli indicatori:

`clearerr(fp); /* azzera gli indicatori di errore e eof di fp */`

D&R

La funzione clearerr non risulta utile molto spesso perché altre funzioni di libreria azzeroano uno o entrambi gli indicatori come side effect.

feof
ferror

D&R

Per controllare gli indicatori di uno stream e determinarne se una delle operazioni precedenti non è andata a buon fine, possiamo chiamare le funzioni feor. La chiamata `feof(fp)` restituisce un valore diverso da zero nel caso in cui l'indicatore end-of-file associato al file `fp` risultasse impostato. La chiamata `ferror(fp)` restituisce un valore diverso da zero nel caso in cui l'indicatore di errore fosse impostato. Entrambe le funzioni restituiscono il valore zero negli altri casi.

Quando la `scanf` restituisce un valore minore di quello che ci si aspettava, possiamo usare le funzioni `feof` e `ferror` per determinarne la ragione. Se la funzione `feof` restituisce un valore diverso da zero significa che abbiamo raggiunto la fine del file. Se la funzione `ferror` restituisce un valore diverso da zero significa che deve essersi verificato un errore di incompatibilità. Indipendentemente da quale fosse stato il problema, il valore restituito dalla `scanf` ci informa di quanti dati siano stati letti prima che l'errore si verificasse.

Per vedere come queste due funzioni possano essere utilizzate, scriviamo una funzione che cerchi all'interno di un file una riga che cominci con un numero intero. Ecco come intendiamo chiamare la funzione

```
n = find_int("foo");
```

"foo" è il nome del file all'interno del quale deve essere effettuata la ricerca. La funzione restituisce il valore dell'intero che viene trovato, il quale viene poi assegnato alla variabile `n`. Se si verifica un problema (il file non può essere aperto, si verifica un errore in lettura oppure nessuna riga inizia con un intero), la funzione `find_int` restituisce un codice di errore (rispettivamente -1, -2 e -3). Assumeremo che nessuna riga inizi con un intero negativo.

```

int find_int(const char *filename)
{
    FILE *fp = fopen(filename, "r");
    int n;

    if (fp == NULL)
        return -1; /* non può aprire il file */

    while (fscanf(fp, "%d", &n) != 1) {
        if (ferror(fp)) {
            fclose(fp);
            return -2; /* errore in lettura */
        }
        if (feof(fp)) {
            fclose(fp);
            return -3; /* intero non trovato */
        }
        fscanf(fp, "%*[^\n]"); /* salta il resto della riga */
    }
    fclose(fp);
    return n;
}

```

L'espressione di controllo del ciclo `while` chiama la `fscanf` cercando di leggere un intero all'interno del file. Se il tentativo non va a buon fine (la `fscanf` restituisce un valore diverso da 1), la `find_int` chiama la `ferror` e la `feof` per capire se il problema era relativo a un errore in lettura o alla fine del file. Se non si è verificato nessuno dei due casi, la `fscanf` deve aver incontrato un errore di incompatibilità e quindi la `find_int` salta la restante parte della riga e prova nuovamente. Fate caso all'uso della conversione `%*[^\n]` per saltare tutti i caratteri fino al prossimo new-line.

22.4 I/O di caratteri

In questa sezione esamineremo le funzioni di libreria che leggono e scrivono singoli caratteri. Queste funzioni lavorano altrettanto bene sia con gli stream di testo che con quelli binari.

Vedremo che le funzioni di questa sezione trattano i caratteri come valori di tipo `int` e non `char`. Una delle ragioni è che le funzioni di input indicano una condizione di end-of-file (o di errore) per mezzo del valore restituito `EOF`, il quale è una costante intera negativa.

Funzioni di output

<code>int fputc(int c, FILE *stream);</code>
<code>int putc(int c, FILE *stream);</code>
<code>int putchar(int c);</code>

`putchar` La funzione `putchar` scrive un carattere nello stream `stdout`:

`fputc(ch); /* scrive ch nello stream stdout */`

`putc` Le funzioni `fputc` e `putc` sono versioni più generali della funzione `putchar` che scrivono un carattere in uno stream qualsiasi:

`fputc(ch, fp); /* scrive ch nello stream fp */`
`putc(ch, fp); /* scrive ch nello stream fp */`

Sebbene `putc` ed `fputc` facciano la stessa cosa, di solito la `putc` viene implementata come una macro (oltre che come una funzione), mentre la `fputc` viene implementata solamente come una funzione. La stessa `putchar` di solito viene implementata come una macro definita in questo modo:

```
#define putchar(c) putc((c), stdout)
```

Può sembrare strano che la libreria fornisca sia la funzione `putc` che la `fputc`. Tuttavia abbiamo visto nella Sezione 14.3 che le macro possono presentare diversi problemi. Lo standard C permette alla macro `putc` di analizzare l'argomento `stream` più di una volta, cosa che alla `fputc` non è permesso fare. Sebbene tipicamente i programmatore preferiscono la `putc` perché rende i programmi più veloci, la funzione `fputc` è disponibile come alternativa.

Se si verifica un errore in scrittura, tutte e tre le funzioni impostano l'indicatore di errore dello stream e restituiscono `EOF`. Se questo non succede, restituiscono il carattere che è stato letto.

Funzioni di input

```
int fgetc(FILE *stream);
int getc(FILE *stream),
int getchar(void),
int ungetc(int c, FILE *stream);
```

`getchar` La funzione `getchar` legge un carattere dallo stream `stdin`:

`ch = getchar(); /* legge un carattere da stdin */`

`fgetc` Le funzioni `fgetc` e `getc` leggono un carattere da uno stream qualsiasi:

`ch = fgetc(fp); /* legge un carattere da fp */`

`ch = getc(fp); /* legge un carattere da fp */`

Tutte e tre le funzioni trattano il carattere come un valore `unsigned char` (che viene poi convertito al tipo `int` prima di essere restituito). Ne risulta che queste non restituiscono mai un valore negativo che non sia `EOF`.

La relazione esistente tra la `getc` e la `fgetc` è simile a quella presente tra la `putc` e la `fputc`. Solitamente la `getc` viene implementata come una macro (oltre che come una funzione), mentre la `fgetc` viene implementata solo sotto forma di funzione. Normalmente anche la `getchar` è una macro:

```
#define getchar() getc(stdin)
```

Per leggere i caratteri da un file, solitamente i programmatore preferiscono la `getc` rispetto alla `fgetc`. Dato che la `getc` normalmente è disponibile sotto forma di macro,

tende a essere più veloce. La fgetc può essere utilizzata come sostituta della getc nel caso questa non fosse appropriata (lo standard permette alla macro getc di analizzare il suo argomento più di una volta e questo potrebbe essere un problema).

Le funzioni fgetc, getc e getchar si comportano allo stesso modo nel caso si verifichasse un problema. Alla fine del file settano l'indicatore end-of-file dello stream e restituiscono EOF. Se si verifica un errore in lettura, queste funzioni settano l'indicatore di errore associato allo stream e restituiscono ancora una volta EOF. Per discriminare tra le due situazioni, possiamo chiamare le funzioni feof e perror.

Uno degli usi più comuni di fgetc, getc e getchar è quello di leggere i caratteri da un file, uno alla volta, fino a quando viene incontrata la fine del file. A tale scopo è comune l'utilizzo del seguente ciclo while:

```
while ((ch = getc(fp)) != EOF) {  
}
```

Dopo aver letto un carattere dal file associato a fp e averlo salvato nella variabile ch (che deve essere di tipo int), il controllo dell'istruzione while confronta ch con EOF. Se ch è diverso da EOF, allora non siamo ancora giunti alla fine del file e quindi viene eseguito il corpo del ciclo. Se ch è uguale a EOF, il ciclo termina.



Il valore restituito da fgetc, getc e getchar va salvato sempre in una variabile int e non in una variabile char. Confrontare una variabile char con EOF può generare il risultato sbagliato.

ungetc

È presente anche un'altra funzione di input per i caratteri, la ungetc, la quale "rimette a posto" un carattere letto dallo stream e azzerà l'indicatore di end-of-file. La capacità di questa funzione può essere comoda nel caso avessimo bisogno di "guardare avanti" di un carattere durante l'input. Per esempio, per leggere una serie di cifre e fermarsi al primo carattere che non sia una cifra, possiamo scrivere [funzione isdigit > 23.5]:

```
while (isdigit(ch = getc(fp))) {  
}  
ungetc(ch, fp); /* rimette a posto l'ultimo carattere letto */
```

Il numero di caratteri che possono essere rimessi a posto da chiamate consecutive della ungetc (senza che intervenga nessuna operazione di lettura) dipende dall'implementazione e dal tipo di stream coinvolto. Si ha garanzia di successo solamente per la prima chiamata. Chiamare una funzione di posizionamento per il file causa la perdita dei caratteri rimessi a posto [funzione di posizionamento per il file > 22.7].

La funzione ungetc restituisce il carattere che le era stato chiesto di rimettere a posto. Tuttavia, restituisce EOF nel caso si tentasse di rimettere a posto EOF o quando si cercasse di rimettere a posto più caratteri di quelli permessi dall'implementazione.

Copiare un file

Il programma seguente effettua la copia di un file. I nomi del file originale e di quello copiato verranno specificati dall'utente sulla riga di comando al momento

dell'esecuzione del programma. Per esempio, per copiare il file f1.c nel file f2.c, useremo il comando
`fcopy f1.c f2.c`

Il programma visualizzerà un messaggio di errore qualora nella riga di comando non venissero immessi esattamente due nomi o nel caso uno dei file non potesse essere aperto.

```
fcopy.c /* Copia un file */

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

int main(int argc, char *argv[])
{
    FILE *source_fp, *dest_fp;
    int ch;

    if (argc != 3) {
        fprintf(stderr, "usage: fcopy source dest\n");
        exit(EXIT_FAILURE);
    }

    if ((source_fp = fopen(argv[1], "rb")) == NULL) {
        fprintf(stderr, "Can't open %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    if ((dest_fp = fopen(argv[2], "wb")) == NULL) {
        fprintf(stderr, "Can't open %s\n", argv[2]);
        fclose(source_fp);
        exit(EXIT_FAILURE);
    }

    while ((ch = getc(source_fp)) != EOF)
        putc(ch, dest_fp);

    fclose(source_fp);
    fclose(dest_fp);
    return 0;
}
```

Utilizzare "rb" e "wb" come modalità di apertura dei file permette al programma fcopy di copiare sia i file testuali che quelli binari. Se al loro posto avessimo utilizzato "r" e "w", il programma non sarebbe stato necessariamente in grado di copiare i file binari.

22.5 I/O di righe

Ora rivolgeremo la nostra attenzione alle funzioni di libreria che leggono e scrivono righe. Queste funzioni vengono utilizzate principalmente con gli stream di testo, sebbene sia possibile usarle anche con gli stream binari.

Funzioni di output

```
int fputs(const char * restrict s,
FILE * restrict stream);
int puts(const char *s);
```

- puts** Nella Sezione 13.3 abbiamo incontrato la funzione puts, la quale scrive una stringa di carattere in stdout:

```
puts("Hi, there!"); /* scrive nello stream stdout */
```

Dopo aver scritto i caratteri all'interno della stringa, la funzione puts aggiunge sempre il carattere new-line.

- fputs** La funzione fputs è una versione più generale della puts. Il suo secondo argomento indica lo stream nel quale deve essere scritto l'output:

```
fputs("Hi, there!", fp); /* scrive in fp */
```

A differenza della puts, la funzione fputs non scrive il carattere new-line a meno che non ne sia presente uno all'interno della stringa.

Entrambe le funzioni restituiscono EOF nel caso si verificasse un errore in scrittura, negli altri casi restituiscono un valore non negativo.

Funzioni di input

```
char *fgets(char * restrict s, int n,
FILE * restrict stream);
char *gets(char *s);
```

- gets** La funzione gets, che abbiamo incontrato per la prima volta nella Sezione 13.3, legge una riga di input da stdin:

```
gets(str); /* legge una riga da stdin */
```

La gets legge i caratteri uno alla volta e li salva nel vettore puntato da str, fino a quando non legge il carattere new-line (che viene scartato).

- fgets** La funzione fgets è una versione più generale della gets che può leggere da qualsiasi stream. La fgets è anche più sicura della gets perché pone un limite al numero di caratteri che andrà a salvare. Ecco come potremmo usare la fgets, assumendo che sia il nome di un vettore di caratteri:

```
fgets(str, sizeof(str), fp); /* legge una riga da fp */
```

Questa chiamata farà sì che la fgets legga i caratteri fino a quando non incontra il primo carattere new-line oppure siano stati letti (str) - 1 caratteri. La prima a verificarsi tra le due possibilità sarà quella che porrà termine alla lettura. Se incontra un carattere new-line, la fgets lo salva assieme agli altri caratteri (quindi: la gets non sa mai il carattere new-line, mentre la fgets lo fa qualche volta).

Sia la gets che la fgets restituiscono un puntatore nullo nel caso si verificasse un errore in lettura o nel caso incontrassero la fine dello stream di input prima di aver salvato qualsiasi carattere (come al solito possiamo chiamare le funzioni feof e ferror per determinare quale situazione si sia verificata). Negli altri casi entrambe restituiscono il loro primo argomento, il quale punta al vettore nel quale è stato salvato l'input. Così come potete aspettarvi, entrambe le funzioni salvano il carattere null alla fine della stringa.

Ora che conosciamo la fgets, possiamo utilizzarla al posto della gets nella maggior parte delle situazioni. Con la gets c'è sempre la possibilità di andare oltre i limiti del vettore di ricezione, di conseguenza il suo utilizzo è sicuro solamente quando abbiamo la certezza che la stringa che si sta leggendo entri perfettamente all'interno del vettore. Quando non ci sono garanzie in tal senso (e solitamente non ce ne sono) è molto più sicuro utilizzare la funzione fgets. Osservate che la fgets legge dallo stream di input standard nel caso le venga passato stdin come suo terzo argomento:

```
fgets(str, sizeof(str), stdin);
```

22.6 I/O di blocchi

```
size_t fread(void * restrict ptr,
            size_t size, size_t nmemb,
            FILE * restrict stream);
size_t fwrite(const void * restrict ptr,
             size_t size, size_t nmemb,
             FILE * restrict stream);
```

D&R
fwrite
Le funzioni fread e fwrite permettono a un programma di leggere e scrivere grossi blocchi di dati in un singolo colpo. Queste funzioni vengono utilizzate principalmente con gli stream binari, sebbene (facendo attenzione) è possibile utilizzarle anche con stream di testo.

La funzione fwrite è stata pensata per copiare un vettore dalla memoria a uno stream. Il primo argomento in una chiamata alla fwrite è costituito dall'indirizzo del vettore, il secondo è la dimensione di ogni elemento (in byte), mentre il terzo rappresenta il numero di elementi che devono essere scritti. Il quarto argomento è un puntatore a file, che indica dove i dati debbano essere scritti. Per esempio, per scrivere l'intero contenuto del vettore a possiamo utilizzare la seguente chiamata alla fwrite:

```
fwrite(a, sizeof(a[0]), sizeof(a) / sizeof(a[0]), fp);
```

Non c'è nessuna regola che ci imponga di scrivere l'intero vettore, possiamo scrivere anche solo una porzione. La fwrite restituisce il numero di elementi (*non byte*) che sono stati effettivamente scritti. Questo numero sarà minore del terzo argomento nel caso si verificasse un errore.

fread La funzione fread legge gli elementi di un vettore da uno stream. I suoi argomenti sono simili a quelli della fwrite: l'indirizzo del vettore, la dimensione di ogni ele-

mento (in byte), il numero di elementi da leggere e un puntatore a file. Per leggere contenuto di un file e salvarlo all'interno del vettore *a*, possiamo utilizzare la seguente chiamata alla *fread*:

```
n = fread(a, sizeof(a[0]), sizeof(a) / sizeof(a[0]), fp);
```

È importante controllare il valore restituito dalla funzione, perché questo indica numero di elementi (*non* byte) che sono stati effettivamente letti. Questo numero dovrebbe essere uguale al terzo argomento, a meno che non sia stata incontrata fine del file di input o si sia verificato un errore in lettura. Le funzioni *feof* e *ferror* possono essere usate per determinare la ragione dell'eventuale carenza.



Fate attenzione a non confondere il secondo e il terzo argomento delle *fread*. Considerate la seguente chiamata alla *fread*:

```
fread(a, 1, 100, fp)
```

Stiamo chiedendo alla *fread* di leggere 100 elementi da un byte, di conseguenza restituirà un valore compreso tra 0 e 100. La chiamata seguente chiede alla *fread* di leggere un blocco di 100 byte:

```
fread(a, 100, 1, fp)
```

In questo caso il valore restituito dalla *fread* sarà uguale a 0 o a 1.

La funzione *fwrite* è conveniente per un programma che ha bisogno di salvare dati in un file prima di terminare. Successivamente, lo stesso programma (o persino un programma diverso) potrà usare la *fread* per leggere i dati e rimetterli nella memoria. A discapito delle apparenze, i dati non hanno bisogno di essere sotto forma vettore. Le funzioni *fwrite* e *fread* funzionano altrettanto bene con variabili di tutti i tipi. Le strutture, in particolare, possono essere lette dalla *fread* e scritte dalla *fwrite*. Per scrivere una variabile struttura in un file, per esempio, possiamo utilizzare la seguente chiamata alla *fwrite*:

```
fwrite(&s, sizeof(s), 1, fp);
```



Fate attenzione quando utilizzate la *fwrite* per scrivere delle strutture che contengono dei puntatori. Non c'è alcuna garanzia che i valori di questi ultimi siano ancora validi dopo la lettura.

22.7 Posizionamento nei file

```
int fgetpos(FILE * restrict stream,
            fpos_t * restrict pos),
int fseek(FILE *stream, long int offset, int whence),
int fsetpos(FILE *stream, const fpos_t *pos),
long int ftell(FILE *stream),
void rewind(FILE *stream);
```

Ogni stream è associato con una **posizione**. Quando un file viene aperto, la posizione del file viene impostata all'inizio del file stesso (se il file viene aperto in modalità accodamento la posizione iniziale può essere sia all'inizio che alla fine del file stesso, dipende dall'implementazione). Successivamente quando viene eseguita un'operazione di lettura o scrittura, la posizione avanza automaticamente e questo ci permette di muoverci all'interno del file in modo sequenziale.

Sebbene un accesso sequenziale vada bene per molte applicazioni, alcuni programmi hanno bisogno di poter effettuare dei salti all'interno del file, accedendo ad alcuni dati in un punto e ad altri dati in un altro punto. Per esempio: se il file contiene una serie di registrazioni, potremmo voler saltare direttamente a una particolare registrazione per leggerla o aggiornarla. L'header <stdio.h> supporta questa forma di accesso fornendo cinque funzioni che permettono a un programma di determinare la posizione del file corrente e di modificarla.

- fseek** La funzione fseek modifica la posizione associata al file indicato dal primo argomento (un puntatore a file). Il terzo argomento specifica se la nuova posizione debba essere calcolata a partire dall'inizio del file, dalla posizione corrente o dalla fine del file. A tale scopo l'header <stdio.h> definisce tre macro:

SEEK_SET	Inizio del file
SEEK_CUR	Posizione corrente
SEEK_END	Fine del file

Il secondo argomento è un conteggio di byte (anche negativo, eventualmente). Per esempio, per spostarsi all'inizio del file, la direzione di ricerca sarà SEEK_SET mentre il conteggio dei byte sarà pari a 0:

```
fseek(fp, 0L, SEEK_SET); /* si sposta all'inizio del file */
```

Per spostarsi alla fine del file la direzione di ricerca sarà SEEK_END:

```
fseek(fp, 0L, SEEK_END); /* si sposta alla fine del file */
```

Per spostarsi indietro di 10 byte, la direzione di ricerca sarà uguale a SEEK_CUR, mentre il conteggio dei byte sarà uguale a -10:

```
fseek(fp, -10L, SEEK_CUR); /* si sposta all'indietro di 10 byte */
```

Osservate che il conteggio dei byte è di tipo long int, per questo motivo abbiamo utilizzato come argomenti 0L e -10L (naturalmente anche 0 e -10 avrebbero funzionato dato che gli argomenti vengono convertiti automaticamente al tipo appropriato).

Normalmente la funzione fseek restituisce uno zero. Se si verifica un errore (per esempio la posizione richiesta non esiste), la funzione restituisce un valore diverso da zero.

In ogni caso le funzioni per il posizionamento dei file vengono utilizzate principalmente con gli stream binari. Il C non proibisce ai programmi di utilizzarle con gli stream di testo, tuttavia in quel caso è necessaria una certa attenzione a causa delle differenze presentate dai diversi sistemi operativi. In particolare la fseek è sensibile al fatto che uno stream sia testuale o binario. Per gli stream testuali si deve avere una delle due condizioni: (1) offset (il secondo argomento della fseek) deve essere uguale a zero, oppure (2) whence (il terzo argomento) deve essere uguale a SEEK_SET e offset deve essere un valore ottenuto da una precedente chiamata alla funzione ftell (in

altre parole, possiamo usare la `fseek` solo per spostarci all'inizio o alla fine dello stream oppure ritornare in un punto che era stato visitato precedentemente). Per gli stream binari alla `fseek` non viene richiesto di supportare chiamate nelle quali il parametro `whence` è uguale a `SEEK_END`.

- ftell** La funzione `ftell` restituisce la posizione del file corrente sotto forma di intero di tipo `long` (se si verifica un errore la `ftell` restituisce `-1L` e salva un codice di errore nel campo `errno` [variabile `errno > 24.21`]). Il valore restituito dalla `ftell` può essere salvato e successivamente passato in una chiamata alla `fseek`, rendendo possibile in questo modo un ritorno a una precedente posizione:

```
long file_pos;
-
file_pos = ftell(fp);           /* salva la posizione corrente */
-
fseek(fp, file_pos, SEEK_SET); /* ritorna nella vecchia posizione */
```

Se `fp` è uno stream binario, la chiamata `ftell(fp)` restituisce la posizione corrente del file sotto forma di un conteggio di byte, dove lo zero rappresenta l'inizio del file. Se invece `fp` è uno stream testuale, la chiamata `ftell(fp)` non è necessariamente un conteggio di byte. Di conseguenza è meglio non eseguire dell'aritmetica sui valori restituiti dalla `ftell`. Per esempio, non è una buona idea sottrarre i valori restituiti da due chiamate `ftell` per vedere quanto sono lontane due posizioni all'interno del file.

- rewind** La funzione `rewind` imposta la posizione del file all'inizio di quest'ultimo. La chiamata `rewind(fp)` è praticamente equivalente alla `fseek(fp, 0L, SEEK_SET)`. La differenza è che `rewind` non restituisce un valore ma azzerà l'indicatore di errore associato a `fp`.

- fgetpos**
fsetpos
D&R Le funzioni `fseek` e `ftell` hanno un problema: sono limitate ai file la cui posizione di lettura può essere contenuta in un intero di tipo `long`. Per lavorare con file molto grandi, la libreria C fornisce due funzioni aggiuntive: la `fgetpos` e la `fsetpos`. Queste funzioni possono gestire file di grandi dimensioni perché per rappresentare le posizioni usano valori del tipo `fpos_t`. Un valore `fpos_t` non è necessariamente un intero, potrebbe essere anche una struttura.

La chiamata `fgetpos(fp, &file_pos)` salva la posizione associata a `fp` nella variabile `file_pos`. La chiamata `fsetpos(fp, &file_pos)` imposta la posizione di `fp` al valore contenuto nella variabile `file_pos` (questo valore deve essere stato ottenuto da una chiamata precedente alla `fgetpos`). Se una chiamata alla `fgetpos` o `fsetpos` non viene eseguita con successo, un codice di errore viene salvato all'interno di `errno`. Entrambe le funzioni restituiscono uno zero quando hanno successo e un valore diverso da zero quando non lo hanno.

Ecco come potremmo utilizzare le funzioni `fgetpos` e `fsetpos` per salvare una posizione di un file e ritornarci in un secondo momento:

```
fpos_t file_pos;
-
fgetpos(fp, &file_pos); /* salva la posizione corrente */
-
fsetpos(fp, &file_pos); /* ritorna alla vecchia posizione */
```

PROGRAMMA

Modificare un file contenente registrazioni di componenti

Il programma seguente apre un file binario contenente delle strutture `part`, le salva in un vettore, imposta al valore zero il membro `on_hand` di tutte le strutture e poi riscrive le strutture nel file. Osservate che il programma apre il file in modalità "rb+", cosa che permette sia la lettura che la scrittura.

```
invclear.c /* Modifica un file contenente registrazioni di componenti impostando
           a zero la quantità disponibile di tutti i componenti */

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

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} inventory[MAX_PARTS];

int num_parts;

int main(void)
{
    FILE *fp;
    int i;

    if ((fp = fopen("inventory.dat", "rb+")) == NULL) {
        fprintf(stderr, "Can't open inventory file\n");
        exit(EXIT_FAILURE);
    }

    num_parts = fread(inventory, sizeof(struct part),
                      MAX_PARTS, fp);

    for (i = 0; i < num_parts; i++)
        inventory[i].on_hand = 0;

    rewind(fp);
    fwrite(inventory, sizeof(struct part), num_parts, fp);
    fclose(fp);

    return 0;
}
```

Chiamare la `rewind` è molto importante. Successivamente alla chiamata alla funzione `fread`, la posizione associata al file si trova alla fine del file stesso. Se chiamassimo la `fwrite` senza prima chiamare la `rewind`, la prima aggiungerebbe dei nuovi dati alla fine del file invece di sovrascrivere quelli vecchi.

22.8 I/O di stringhe

Le funzioni descritte in questa sezione sono un po' inusuali, dato che non hanno nulla a che fare con gli stream o i file. Permettono invece di leggere e scrivere dati utilizzando una stringa come se fosse uno stream. Le funzioni `sprintf` e `snprintf` scrivono dei caratteri nello stesso modo con il quale verrebbero scritti in uno stream. La funzione `sscanf` legge dei caratteri da una stringa esattamente come se venissero letti da uno stream. Queste funzioni, che somigliano molto alla `printf` e alla `scanf`, sono piuttosto utili. La `sprintf` e la `snprintf` ci danno accesso alle capacità di formattazione dei dati della `printf` senza dover scrivere i dati in uno stream. Analogamente la `sscanf` ci dà accesso alle potenti capacità di pattern-matching della `scanf`. La parte rimanente di questa sezione tratta nel dettaglio le funzioni `sprintf`, `snprintf` e `sscanf`.

All'header `<stdio.h>` appartengono anche tre funzioni simili: la `vfprintf`, la `vsnprintf` e la `vsscanf`. Tuttavia queste funzioni si basano sul tipo `va_list`, che viene dichiarato nell'header `<stdarg.h>`; rimandiamo la loro trattazione alla Sezione 26.1.

Funzioni di output

```
int sprintf(char * restrict s,
           const char * restrict format, ...);
int snprintf(char * restrict s, size_t n,
             const char * restrict format, ...);
```

Nota: In questo e nei capitoli seguenti, i prototipi di funzione che sono stati introdotti dal C99 verranno scritti in corsivo. Anche il nome delle funzioni verrà scritto in corsivo quando compare nel margine sinistro.

sprintf La funzione `sprintf` è simile alla `printf` e alla `fprintf`, ma a differenza di queste scrive il suo output in un vettore di caratteri (puntato dal suo primo argomento) invece che in uno stream. Il secondo argomento della `sprintf` è una stringa di formato identica a quella utilizzata dalla `printf` e dalla `fprintf`. Per esempio, la chiamata

```
sprintf(date, "%d/%d/%d", 9, 20, 2010);
```

scrivrà "9/20/2010" all'interno della stringa `date`. Quando ha terminato la scrittura, la `sprintf` aggiunge il carattere null e restituisce il numero di caratteri salvati (senza contare il carattere null). Se si verifica un errore di codifica (un wide character non può essere tradotto in un carattere multibyte valido), la `sprintf` restituisce un valore negativo.

La `sprintf` ha un gran numero di applicazioni. Per esempio, occasionalmente potremmo voler formattare i dati dell'output senza scriverli effettivamente. Possiamo utilizzare la `sprintf` per fare la formattazione e poi salvare il risultato in una stringa fino a quando non viene il momento di produrre l'output. La `sprintf` è comoda anche per convertire i numeri nel formato a caratteri.

snprintf La funzione `snprintf` è uguale alla `sprintf` a eccezione del fatto che possiede il parametro aggiuntivo `n`. Nella stringa non verranno scritti più di `n - 1` caratteri, senza contare il carattere null di termine che viene sempre scritto a meno che `n` non sia uguale a zero (quindi possiamo dire che la `snprintf` scrive nella stringa `n` caratteri, l'ultimo dei quali è il carattere null). Per esempio, la chiamata

```
snprintf(name, 13, "%s, %s", "Einstein", "Albert");
```

scriverrà "Einstein, Al" all'interno della stringa name.

La snprintf restituisce il numero di caratteri che verrebbero scritti (escluso il carattere null) nel caso in cui non ci fosse restrizione sulla lunghezza. Se si verifica un errore di codifica, la snprintf restituisce un numero negativo. Per vedere se la snprintf ha avuto spazio a sufficienza per scrivere tutti i caratteri richiesti, possiamo controllare se il valore restituito non è negativo e se è minore di n.

Funzioni di input

```
int sscanf(const char * restrict s,
           const char * restrict format, ...);
```

sscanf La funzione sscanf è simile alle funzioni scanf e fscanf, ma differisce da queste per il fatto che legge da una stringa (puntata dal suo primo argomento) invece che da uno stream. Il secondo argomento della sscanf è una stringa di formato identica a quella usata dalla scanf e dalla fscanf.

La funzione sscanf è comoda per estrarre dati da una stringa che è stata letta con un'altra funzione di input. Per esempio, potremmo utilizzare la fgets per ottenere una riga di input e poi passarla alla sscanf per ulteriori elaborazioni:

```
fgets(str, sizeof(str), stdin); /* legge una riga di input */
sscanf(str, "%d%d", &i, &j); /* estrae due interi */
```

Un vantaggio dell'uso della sscanf al posto della scanf o della fscanf è che possiamo esaminare una riga di input tutte le volte che è necessario e non solamente una. Questo facilita il riconoscimento di formati alternativi di input e la ripresa dagli errori. Considerate il problema di leggere una data che è stata scritta o nel formato *mese/giorno/anno* o nel formato *mese-giorno-anno*. Assumendo che str contenga una riga di input, possiamo estrarre il mese, il giorno e l'anno nel modo seguente:

```
if (sscanf(str, "%d /%d /%d", &month, &day, &year) == 3)
    printf("Month: %d, day: %d, year: %d\n", month, day, year);
else if (sscanf(str, "%d -%d -%d", &month, &day, &year) == 3)
    printf("Month: %d, day: %d, year: %d\n", month, day, year);
else
    printf("Date not in the proper form\n");
```

Come le funzioni scanf e fscanf, anche la sscanf restituisce il numero di dati letti con successo e salvati. La sscanf restituisce il valore EOF se raggiunge la fine della stringa (segnata dal carattere null) prima di trovare il primo dato.

Domande & Risposte

D: Se utilizziamo il reindirizzamento dell'input o dell'output, i nomi dei file reindirizzati compaiono come argomenti della riga di comando? [p. 557]

R: No, il sistema operativo li rimuove dalla riga di comando. Supponiamo di far girare un programma immettendo il comando

`demo foo <in_file bar >out_file baz`

Il valore di argc sarà pari a 4, argv[0] punterà al nome del programma, argv[1] punterà a "foo", argv[2] punterà a "bar" e argv[3] punterà a "baz".

D: Pensavamo che la fine di una riga fosse sempre segnalata da un carattere new-line. Ora stiamo dicendo che il marcatore end-of-line varia a seconda del sistema operativo. Come si spiega questa discrepanza? [p. 558]

R: Le funzioni della libreria del C fanno *sembrare* che ogni riga termini con un singolo carattere new-line. Indipendentemente dal fatto che il file di input contenga un carattere carriage-return, un carattere line-feed o entrambi, le funzioni di libreria come la getc restituiscono un unico carattere new-line. Le funzioni di output eseguono la traduzione inversa. Se un programma chiama una funzione di libreria per scrivere il carattere new-line in un file, la funzione tradurrà il carattere nel marcatore end-of-line appropriato. L'approccio del C rende i programmi più portabili e più facili da scrivere. Possiamo lavorare con i file di testo senza doverci preoccupare di come venga effettivamente rappresentato il marcatore end-of-line. Osservate che l'input/output eseguito su un file aperto in modalità binaria non è soggetto a nessuna traduzione di caratteri (carriage-return e line-feed sono trattati come tutti gli altri caratteri).

D: Stiamo scrivendo un programma che deve salvare dei dati all'interno di un file in modo che successivamente possano essere letti da un altro programma. È meglio salvare i dati in forma testuale o binaria? [p. 558]

R: Dipende. Se i dati sono costituiti interamente da testo non c'è alcuna differenza. Se invece i dati contengono numeri, allora la decisione è più complessa.

Solitamente la forma binaria è quella preferibile visto che può essere letta e scritta velocemente. I numeri sono già in forma binaria quando sono immagazzinati nella memoria e quindi copiarli all'interno di un file è facile. Scrivere numeri nella forma testuale è molto più lento visto che ogni numero deve essere convertito (di solito dalla `fprintf`) nella forma a caratteri. Leggere il file in un secondo momento richiede a sua volta del tempo visto che i numeri devono essere convertiti nuovamente in forma binaria. Inoltre, come abbiamo visto nella Sezione 22.1, spesso salvare dei dati in forma binaria ci permette di risparmiare spazio.

I file binari, tuttavia, presentano due inconvenienti. Sono difficili da leggere per gli esseri umani e questo può intralciare le operazioni di debugging. Inoltre, generalmente i file binari non sono portabili da un sistema all'altro, visto che tipi diversi di computer possono salvare i dati in modi diversi. Alcune macchine, per esempio, salvano i valori int usando due byte mentre altre usano quattro byte. C'è anche la questione dell'ordinamento dei byte (big-endian contro little-endian).

D: I programmi C per il sistema UNIX non sembrano usare la lettera b nella stringa di modalità anche quando i file che vengono aperti sono binari. Perché? [p. 561]

R: In UNIX i file testuali e quelli binari hanno esattamente lo stesso formato e quindi non c'è bisogno di utilizzare la lettera b. I programmati UNIX, però, dovrebbero comunque includere la lettera b in modo che i loro programmi siano più portabili.

D: Abbiamo visto programmi che chiamano la fopen e mettono la lettera t nella stringa di modalità. Cosa significa?

R: Lo standard C ammette che nella stringa di modalità compaiano dei caratteri aggiuntivi, ammesso che questi seguano i caratteri r, w, a, b e +. Alcuni compilatori ammettono l'utilizzo della lettera t per indicare che il file viene aperto in modalità testuale invece che binaria. Naturalmente la modalità testuale è in ogni caso quella di default, quindi la t non aggiunge nulla. Quando possibile è meglio evitare l'utilizzo della lettera t e delle altre caratteristiche non portabili.

D: Perché preoccuparsi di chiamare la fclose per chiudere un file? Non è forse vero che tutti i file aperti vengono chiusi automaticamente quando il programma termina? [p. 561]

R: Di solito questo è vero, ma non se il programma chiama la funzione abort [funzione abort > 26.2] per terminare. Anche quando la funzione abort non viene utilizzata, ci sono comunque delle buone ragioni per chiamare la fclose. Per prima cosa questo riduce il numero di file aperti. I sistemi operativi pongono un limite al numero di file che un programma può mantenere aperti contemporaneamente. I programmi di grosse dimensioni possono scontrarsi con questo limite (la macro FOPEN_MAX definita in <stdio.h> specifica il numero minimo di file che l'implementazione garantisce che possano essere aperti simultaneamente). In secondo luogo, il programma diventa più facile da capire e modificare. Cercando la chiamata fclose, il lettore può determinare il punto dal quale un file non viene più utilizzato. Il terzo motivo riguarda la sicurezza. Chiudere un file assicura che il suo contenuto e la sua directory siano aggiornati correttamente. Se il programma dovesse andare in crash, almeno il file sarebbe intatto.

D: Stiamo scrivendo un programma che chiederà all'utente di immettere il nome di un file. Quanto lungo dovrà essere il vettore che conterrà tale nome? [p. 563]

R: Dipende dal vostro sistema operativo. Fortunatamente, per specificare la dimensione del vettore, potete utilizzare la macro FILENAME_MAX (definita in <stdio.h>). Questa macro rappresenta la lunghezza della stringa contenente il nome più lungo per un file che l'implementazione garantisce si possa aprire.

D: La funzione fflush può svuotare uno stream che era stato aperto sia in lettura che scrittura? [p. 566]

R: Secondo lo standard C, l'effetto di chiamare fflush è definito per uno stream che: (a) era stato aperto per l'output oppure (b) era stato aperto per l'aggiornamento e la cui ultima operazione non sia stata una lettura. In tutti gli altri casi l'effetto di una chiamata alla fflush non è definito. Quando alla fflush viene passato un puntatore nullo, questa svuota tutti gli stream che soddisfano la condizione (a) o la condizione (b).

D: La stringa di formato di una chiamata ...printf o ...scanf può essere costituita da variabile?

R: Sì, può essere qualsiasi espressione del tipo char *. Questa proprietà rende le funzioni ...printf e ...scanf ancora più versatili di quello che ci potevamo aspettare. Considerate l'esempio classico tratto dal libro *The C Programming Language* di Kernighan e Ritchie, il quale stampa gli argomenti della riga di comando separati da spazi:

```
while (--argc > 0)
    printf((argc > 1) ? "%s " : "%s", *++argv);
```

La stringa di formato è rappresentata dall'espressione (`argc > 1`) ? `"%s"` : `"%s"`, la quale restituisce la stringa `"%s"` per tutti gli argomenti della riga di comando eccetto l'ultimo.

D: Tra le funzioni di libreria quali, a parte la `clearerr`, azzerano gli indicatori di errore e di end-of-file di uno stream? [p. 583]

R: Chiamare la funzione `rewind`, mentre apre o riapre lo stream, azzerà entrambi gli indicatori. Chiamare le funzioni `ungetc`, `fseek` o `fsetpos`, invece, azzerà solo l'indicatore di end-of-file.

D: Non riusciamo a far funzionare la `feof`, sembra che restituiscia uno zero anche alla fine del file. Che cosa stiamo sbagliando? [p. 583]

R: La `feof` restituisce un valore diverso da zero solo quando una precedente operazione di lettura non è andata a buon fine. Non potete usare la `feof` per controllare la fine del file *prima* di cercare di leggere. Dovete prima cercare di leggere e poi controllare il valore restituito dalla funzione. Se il valore restituito indica che l'operazione non è andata a buon fine, allora potete utilizzare la funzione per determinare se l'evento è stato causato dal raggiungimento della fine del file. In altre parole, non è il massimo pensare alla funzione `feof` come a un modo per *rilevare* la fine del file. Pensatela invece come a un modo per *confermare* che la fine del file era la ragione del fallimento dell'operazione di lettura.

D: Ancora non capiamo perché la libreria di I/O fornisce delle macro chiamate `putc` e `getc` in aggiunta alle funzioni chiamate `fputc` e `fgetc`. Secondo la Sezione 21.1 ci sono già due versioni di `putc` e `getc` (una macro e una funzione). Se abbiamo bisogno di una vera funzione invece di una macro, possiamo esporre le funzioni `putc` e `getc` cancellando la definizione delle macro. Quindi perché esistono `fputc` e `fgetc`? [p. 585]

R: Ragioni storiche. Prima della standardizzazione, il C non aveva nessuna regola che garantisse l'esistenza di vere funzioni dietro a ogni macro parametrica presente nella libreria. Tradizionalmente `putc` e `getc` venivano implementate solo come macro, mentre `fputc` e `fgetc` venivano implementate come funzioni.

***D:** Perché è sbagliato salvare il valore restituito dalle funzioni `fgetc`, `getc` o `getchar` in una variabile `char`? Non capiamo perché confrontare una variabile `char` con `EOF` possa generare il risultato scorretto. [p. 586]

R: Ci sono due casi nei quali questo confronto può dare il risultato sbagliato. Per rendere concreta questa discussione assumeremo di utilizzare l'aritmetica in complemento a due.

Per prima cosa supponete che il tipo `char` sia senza segno (ricordate che alcuni compilatori trattano `char` come un tipo con segno, mentre altri lo trattano come un tipo senza segno). Supponete ora che la `getc` restituisca `EOF` e di memorizzarlo in una variabile `char` chiamata `ch`. Se `EOF` rappresenta `-1` (il suo valore tipico), `ch` si troverà a possedere il valore `255`. Confrontare `ch` (un carattere senza segno) con `EOF` (un intero con segno) impone la conversione di `ch` in un intero con segno (`255` in questo caso). Il confronto con `EOF` ha esito negativo dato che `255` è diverso da `-1`.

Assumete ora che `char` sia un tipo con segno. Considerate quello che succede nel caso la `getc` legga da uno stream binario un byte contenente il valore `255`. Salvare `255`

nella variabile ch le assegna il valore -1 dato che quest'ultima è un carattere con segno. Controllare se ch è uguale a EOF restituirà erroneamente il valore true.

D: Le funzioni per l'input dei caratteri descritte nella Sezione 22.4, prima di poter leggere quanto digitato dall'utente, richiedono che venga premuto il tasto Invio. Com'è possibile scrivere un programma che risponda a pressioni individuali di tasti?

R: Come avete notato le funzioni getc, fgetc e getchar sono sottoposte a buffering. Non iniziano a leggere l'input fino a quando l'utente non ha premuto il tasto Invio. Per leggere i caratteri immessi (che è importante per alcuni programmi) avrete bisogno di usare una libreria non standard fornita con il vostro sistema operativo. In UNIX, per esempio, la libreria curses di solito fornisce questa possibilità.

D: Quando stiamo leggendo l'input dell'utente, come possiamo saltare tutti i caratteri lasciati nella riga di input corrente?

R: Una possibilità è quella di scrivere una piccola funzione che legga e ignori tutti i caratteri fino al primo carattere new-line (quest'ultimo incluso):

```
void skip_line(void)
{
    while (getchar() != '\n')
        ;
}
```

Un'altra possibilità è quella di chiedere alla scanf di saltare tutti i caratteri fino al primo carattere new-line:

```
scanf("%*[^\n]"); /* salta tutti i caratteri fino a new-line */
```

La scanf leggerà tutti i caratteri fino al primo carattere new-line, ma non li salverà in nessun luogo (il carattere * indica la soppressione dell'assegnamento). L'unico problema con l'uso della scanf è che questa lascia non letto il carattere new-line e quindi c'è bisogno di scartarlo separatamente.

Qualsiasi cosa facciate non chiamate la funzione fflush:

```
fflush(stdin); /* l'effetto è indefinito */
```

Sebbene alcune implementazioni permettano di utilizzare la fflush per svuotare l'input non letto, non è una buona idea assumere che lo facciano tutte. La fflush è pensata per svuotare gli stream di output. Lo standard C asserisce che il suo effetto sugli stream di input è indefinito.

D: Perché non è una buona idea utilizzare le funzioni fread e fwrite con gli stream testuali? [p. 589]

R: Una difficoltà consiste nel fatto che, con alcuni sistemi operativi, il carattere new-line diventa una coppia di caratteri quando viene scritto su un file di testo (si veda la Sezione 22.1 avere per maggiori dettagli). Dobbiamo tenere conto di questa espansione, altrimenti perderemo traccia dei nostri dati. Per esempio, se utilizzassimo la fwrite per scrivere blocchi di 80 caratteri, alcuni dei blocchi potrebbero finire per occupare più di 80 byte a causa del fatto che i caratteri new-line sono stati espansi.

D: Perché sono presenti due insiemi di funzioni per il posizionamento (fseek/ftell e fsetpos/fgetpos)? Uno dei due non sarebbe stato sufficiente? [p. 592]

R: Le funzioni fseek e ftell fanno parte della libreria del C da un'eternità, però hanno un inconveniente: assumono che una posizione in un file entri in un valore long int. Dato che tipicamente un long int è un tipo a 32 bit, questo significa che queste due funzioni non possono lavorare con file contenenti più di 2.147.483.647 byte. Riconoscendo questo problema, quando è stato creato il C89, le funzioni fsetpos e fgetpos sono state aggiunte all'header <stdio.h>. A queste funzioni non è richiesto di trattare le posizioni come numeri e di conseguenza non sono soggette alle restrizioni del tipo long int. Non pensate però di essere costretti a utilizzare la fsetpos e la fgetpos: se la vostra implementazione supporta il tipo long int a 64 bit, queste due funzioni vanno bene per file veramente grandi.

D: Perché questo capitolo non tratta del controllo dello schermo, ovvero muovere il cursore, modificare il colore dei caratteri sullo schermo e così via?

R: Il C non prevede delle funzioni standard per il controllo dello schermo. Lo standard C si occupa solo di questioni che possono essere ragionevolmente standardizzate su una grande varietà di computer e sistemi operativi, il controllo dello schermo è al di fuori da tutto questo. Il modo peculiare per risolvere questi problemi con il sistema UNIX è quello di utilizzare la libreria curses che supporta il controllo dello schermo in modo indipendente dal terminale.

Analogamente non sono presenti funzioni per creare programmi provvisti di interfaccia grafica. Tuttavia molto probabilmente potrete utilizzare chiamate a funzioni C per accedere all'API (*Application Programming Interface*) di programmazione a finestre per il vostro sistema operativo.

Esercizi

Sezione 22.1

- Indicate se i seguenti file contengono con maggiore probabilità dei dati testuali o dei dati binari.
 - Un file di codice oggetto prodotto da un compilatore C.
 - Un programma prodotto da un compilatore C.
 - Un messaggio e-mail inviato da un computer a un altro.
 - Un file contenente un'immagine grafica.

Sezione 22.2



- Indicate quale stringa di modalità è più probabile che venga passata alla funzione fopen in ognuna delle situazioni seguenti.
 - Un sistema di gestione di un database apre un file contenente dei record che devono essere aggiornati.
 - Un programma di posta apre un file contenente i messaggi salvati in modo da poter aggiungerne di ulteriori alla fine.
 - Un programma grafico apre un file contenente un'immagine che deve essere visualizzata sullo schermo.

- (d) Un interprete di comandi di un sistema operativo apre uno "script di shell" (o un "file batch") contenente comandi che devono essere eseguiti.
3. Trovate l'errore presente nel frammento di programma riportato di seguito e indicate come correggerlo.

```
FILE *fp;
if (fp = fopen(filename, "r")) {
    leggere i caratteri fino alla fine del file
}
fclose(fp);
```

- Sezione 22.3** 4. Mostrate come i seguenti numeri verrebbero visualizzati dalla funzione printf con la specifica di conversione %#012.5g:

- (a) 83.7361
 - (b) 29748.6607
 - (c) 1054932234.0
 - (d) 0.0000235218
5. Con la funzione printf c'è qualche differenza tra la specifica di conversione %.4d e la %04d? In caso affermativo spiegate in cosa consiste.

6. *Scrivete una chiamata alla funzione printf che stampi

1 widget

nel caso in cui la variabile widget (di tipo int) avesse il valore 1, e
n widgets

altrimenti, dove n è il valore posseduto da widget. Non vi è permesso utilizzare l'istruzione if o qualsiasi altra istruzione. La risposta dovrà consistere in una singola chiamata alla printf.

7. *Supponete di chiamare la scanf nel modo seguente:

n = scanf("%d%f%d", &i, &x, &j);

(i, j ed n sono variabili int, x è una variabile float). Assumendo che lo stream di input contenga i caratteri mostrati, fornite i valori di i, j, n e x dopo la chiamata. Indicate inoltre quali caratteri vengono consumati dalla chiamata.

- (a) 10•20•30•
- (b) 1.0•2.0•3.0•
- (c) 0.1•0.2•0.3•
- (d) a.1•.2•.3•

8. Nei capitoli precedenti, quando volevamo saltare i caratteri di spazio bianco e leggere i caratteri non bianchi, per la scanf abbiamo utilizzato la stringa di for-

mato " %c". Alcuni programmati utilizzano al suo posto la stringa "%is". Le due tecniche sono equivalenti? In caso contrario quali sono le differenze?

- Sezione 22.4**
9. Quale delle seguenti chiamate *non* è valida per leggere un carattere dallo standard input?
 - (a) getch()
 - (b) getchar()
 - (c) getc(stdin)
 - (d) fgetc(stdin)
 10. Il programma fcopy.c ha un piccolo difetto: quando va a scrivere sul file di destinazione non controlla se si verificano errori. Gli errori durante la scrittura sono rari, tuttavia a volte si verificano (il disco potrebbe diventare pieno, per esempio). Illustrate come aggiungere al programma il mancante controllo di errore. Assumete di voler stampare un messaggio e di terminare immediatamente il programma nel caso si verificasse un errore.
 11. Il seguente ciclo compare nel programma fcopy.c:

```
while ((ch = getc(source_fp)) != EOF)
    putc(ch, dest_fp);
```

Supponete di aver dimenticato di mettere le parentesi attorno a ch = getc(source_fp):

```
while (ch = getc(source_fp) != EOF)
    putc(ch, dest_fp);
```

Il programma compilerà senza errori? Se sì, cosa farà il programma durante l'esecuzione?

12. Trovate l'errore presente nella seguente funzione e mostrate come correggerlo.

```
int count_periods(const char *filename)
{
    FILE *fp;
    int n = 0;
    if ((fp = fopen(filename, "r")) != NULL) {
        while (fgetc(fp) != EOF)
            if (fgetc(fp) == '.')
                n++;
        fclose(fp);
    }
    return n;
}
```

13. Scrivete la seguente funzione:

```
int line_length(const char *filename, int n);
```

La funzione dovrà restituire la lunghezza della riga *n* presente nel file di testo il cui nome corrisponde a *filename* (assumendo che la prima riga del file sia la riga numero 1). Se la riga non esiste, la funzione deve restituire il valore 0.

Sezione 22.5

W

14. (a) Scrivete una vostra versione della funzione *fgets*. Fate in modo che si comporti il più possibile come la vera funzione *fgets*. In particolare, assicuratevi che presenti il corretto valore restituito. Per evitare conflitti con la libreria standard non chiamate la vostra funzione *fgets*.

- (b) Scrivete una vostra versione della *fputs* seguendo le stesse regole del punto (a).

Sezione 22.7

W

15. Scrivete delle chiamate alla *fseek* che eseguano le seguenti operazioni di posizionamento su un file binario i cui dati sono arrangiati in record da 64 byte. Usate *fp* come puntatore a file in tutti i casi.

- (a) Posizionatevi all'inizio del record *n* (assumete che il primo record presente nel file sia il record numero 0).
- (b) Posizionatevi all'inizio dell'ultimo record del file.
- (c) Spostatevi in avanti di un record.
- (d) Spostatevi indietro di due record.

Sezione 22.8

16. Assumete che *str* sia una stringa contenente un "sale rank" immediatamente preceduto dal simbolo # (altri caratteri possono precedere il carattere # e/o seguire il sale rank). Un sale rank è costituito da una serie di cifre decimali che possono contenere delle virgole. Ecco alcuni esempi:

989
24,675
1,162,620

Scrivete una chiamata alla *sscanf* che estragga il sale rank (ma non il simbolo #) e lo salvi in una variabile chiamata *sales_rank*.

Progetti di programmazione

1. Estendete il programma *canopen.c* della Sezione 22.2 in modo che l'utente possa mettere sulla riga di comando un qualsiasi numero di nomi file:

canopen foo bar baz

Il programma dovrà stampare separatamente per ogni file il messaggio can be opened o can't be opened. Fate in modo che il programma termini con lo stato *EXIT_FAILURE* se tra i file uno o più non possono essere aperti.

W

2. Scrivete un programma che converta in maiuscole tutte le lettere presenti in un file (i caratteri che non rappresentano delle lettere non dovranno essere modificati). Il programma deve ottenere il nome del file dalla riga di comando e scrivere il suo output su *stdout*.

3. Scrivete un programma chiamato fcat che esegua il concatenamento di un numero qualsiasi di file scrivendoli uno dopo l'altro nello standard output. Tra i file non dovranno essere lasciati spazi. Il comando seguente, per esempio, visualizzerà sullo schermo i file f1.c, f2.c e f3.c:

```
fcat f1.c f2.c f3.c
```

fcat deve generare un messaggio di errore nel caso uno dei file non potesse essere aperto. Suggerimento: dato che non ha mai più di un file aperto alla volta, fcat ha bisogno di una sola variabile puntatore a file. Una volta che ha finito con un file il programma può utilizzare la stessa variabile per aprire quello successivo.

- W 4. (a) Scrivete un programma che conti il numero di caratteri contenuti in un file di testo.
- (b) Scrivete un programma che conti il numero di parole contenute in un file di testo (per "parola" si intende una qualsiasi sequenza di caratteri che non rappresenti dello spazio bianco).
- (c) Scrivete un programma che conti il numero di righe contenute in un file di testo.
Fate in modo che ogni programma ottenga il nome del file dalla riga di comando.
5. Il programma xor.c della Sezione 20.1 rifiuta di tradurre i byte che (nella forma originale o in quella cifrata) corrispondono a caratteri di controllo. Ora possiamo rimuovere questa restrizione. Modificate il programma in modo che i nomi dei file di input e di output siano argomenti della riga di comando. Aprite entrambi i file in modalità binaria e rimuovete il test che controlla se i caratteri originali o quelli cifrati sono stampabili.
- W 6. Scrivete un programma che visualizzi il contenuto del file sotto forma di byte e di caratteri. Fate in modo che l'utente specifichi il nome del file sulla riga di comando. Ecco come dovrà presentarsi l'output del programma nel caso venisse usato per visualizzare il file pun.c della Sezione 2.1:

Offset	Bytes	Characters
0	23 69 6E 63 6C 75 64 65 20 3C	#include <
10	73 74 64 69 6F 2E 68 3E 0D 0A	stdio.h>..
20	0D 0A 69 6E 74 20 6D 61 69 6E	..int main
30	28 76 6F 69 64 29 0D 0A 7B 0D	(void)..{.
40	0A 20 20 70 72 69 6E 74 66 28	. printf(
50	22 54 6F 20 43 2C 20 6F 72 20	"To C, or
60	6E 6F 74 20 74 6F 20 43 3A 20	not to C:
70	74 68 61 74 20 69 73 20 74 68	that is th
80	65 20 71 75 65 73 74 69 6F 6E	e question
90	2E 5C 6E 22 29 3B 0D 0A 20 20	.\n");..
100	72 65 74 75 72 6E 20 30 3B 0D	return 0;.
110	0A 7D	.}

Ogni riga mostra 10 byte del file sotto forma di numeri esadecimale e di caratteri. Il numero presente nella colonna Offset indica la posizione all'interno del file del primo byte della riga. Vengono visualizzati solo i caratteri stampabili (come indicato dalla funzione `isprint`), gli altri caratteri vengono rappresentati con punti. Osservate che l'aspetto di un file testuale può variare a seconda del set di caratteri e del sistema operativo. L'esempio precedente assume che `pun.c` sia un file Windows e quindi i byte 0D e 0A (i codici ASCII per carriage-return e line-feed) sono presenti alla fine di ogni riga. Suggerimento: assicuratevi di aprire il file in modalità "rb".

7. Delle diverse tecniche per comprimere un file, una delle più semplici e veloci è conosciuta come **run-length encoding**. Questa tecnica comprime un file sostituendo le sequenze di byte identici con una coppia di byte: il conto del numero delle ripetizioni seguito dal byte che deve essere ripetuto. Per esempio, supponete che il file che deve essere compresso inizi con la seguente sequenza di byte (mostrata in esadecimale):

```
46 6F 6F 20 62 61 72 21 21 21 20 20 20 20 20
```

Il file compresso conterrà i byte seguenti:

```
01 46 02 6F 01 20 01 62 01 61 01 72 03 21 05 20
```

La tecnica del run-length encoding funziona bene se il file originale contiene molte sequenze di byte identici che abbiano una certa lunghezza. Nel caso peggiore (un file senza byte ripetuti), la tecnica raddoppia la dimensione del file.

- (a) Scrivete un programma chiamato `compress_file` che utilizzi la tecnica del run-length encoding per comprimere un file. Per eseguire il programma dovremo utilizzare un comando del tipo

```
compress_file file-originale
```

Il programma scriverà la versione compressa di `file-originale` in un file chiamato `file-originale.rle`. Per esempio, il comando

```
compress_file foo.txt
```

farà in modo che il programma scriva una versione compressa del file `foo.txt` all'interno di un file chiamato `foo.rle`. Suggerimento: il programma descritto nel Progetto di programmazione 6 può essere utile per il debugging.

- (b) Scrivete un programma chiamato `uncompress_file` che inverta la compressione effettuata dal programma `compress_file`. Il comando `uncompress_file` avrà la seguente forma:

```
uncompress_file file-compresso
```

il file compresso deve avere l'estensione `.rle`. Per esempio il comando

```
uncompress_file foo.txt.rle
```

farà in modo che il programma apra il file `foo.txt.rle` e scriva una versione scompattata del suo contenuto nel file `foo.txt`. Il programma deve visualizzare un

messaggio di errore nel caso il suo argomento della riga di comando non finisca con l'estensione .rle.

8. Modificate il programma `inventory.c` della Sezione 16.3 aggiungendo a questo due nuove operazioni:
 - salvare il database in uno specifico file;
 - caricare il database da un particolare file.

Per rappresentare queste operazioni utilizzate rispettivamente i codici `d` (dump) e `r` (restore). L'interazione con l'utente deve avere questo aspetto:

Enter operation code: d
 Enter name of output file: inventory.dat

Enter operation code: r
 Enter name of input file: inventory.dat

Suggerimento: utilizzate la funzione `fwrite` per scrivere il vettore contenente i componenti in un file binario. Utilizzate la funzione `fread` per recuperare il vettore leggendolo da file.

9. Scrivete un programma che fonda due file contenenti dei record di componenti ottenuti dal programma `inventory.c` (si veda il Progetto di Programmazione 8). Assumete che i record presenti in ogni file siano ordinati per numero di componente e che si voglia che il file risultante sia a sua volta ordinato. Se entrambi i file contengono un componente con lo stesso numero, le quantità salvate nei due record devono essere combinate (come controllo di consistenza fate in modo che il programma confronti i nomi dei componenti e stampi un messaggio di errore se questi non corrispondono). Fate in modo che il programma ottenga i nomi dei file di input e del file di output dalla riga di comando.
10. *Modificate il programma `inventory2.c` della Sezione 17.5 aggiungendo le operazioni `d` (dump) e `r` (restore) descritte nel Progetto di programmazione 8. Dato che le strutture dei componenti non sono contenute in un vettore, l'operazione `d` non può salvarle tutte con una singola chiamata alla `fwrite`. Avrà bisogno invece di visitare ogni nodo della lista concatenata, scrivendo il numero del componente, il suo nome e la quantità disponibile (non salvate il puntatore `next` perché non sarà più valido dopo che il programma ha termine). Quando legge i componenti dal file, l'operazione `r` dovrà ricreare la lista un nodo alla volta.
11. Scrivete un programma che legga una data dalla riga di comando e la visualizzi nel seguente formato:

September 13, 2010

Permettete all'utente di immettere le date sia come 9-13-2010 che come 9/13/2010. Potete assumere che all'interno di una data non siano presenti spazi. Stampate un messaggio di errore se la data non è in uno dei formati specificati. Suggerimento:

usate la `sscanf` per estrarre il mese, il giorno e l'anno dall'argomento della riga di comando.

12. Modificate il Progetto di Programmazione 2 del Capitolo 3 in modo che il programma legga da un file una serie di articoli e visualizzi i dati in una colonna. Ogni riga del file deve presentarsi in questo modo:

item,price,mm/dd/yyyy

Supponete, per esempio, che il file contenga le seguenti righe:

583,13.5,10/24/2005
3912,599.99,7/27/2008

L'output del programma deve avere il seguente aspetto:

Item	Unit	Purchase
	Price	Date
583	\$ 13.50	10/24/2005
3912	\$ 599.99	7/27/2008

Fate in modo che il programma ottenga il nome del file dalla riga di comando.

13. Modificate il Progetto di programmazione 8 del Capitolo 5 in modo che il programma ottenga gli orari di partenza e di arrivo da un file chiamato `flights.dat`. Ogni riga del file contiene un orario di partenza seguito da un orario di arrivo, con uno o più spazi a separare i due. Gli orari devono essere espressi utilizzando un orologio a 24 ore. Ecco un esempio di come dovrebbe presentarsi il file `flights.dat` se contenesse le informazioni sui voli elencate nel progetto originale:

8:00 10:16
9:43 11:52
11:19 13:31
12:47 15:00
14:00 16:08
15:45 17:55
19:00 21:20
21:45 23:58

14. Modificate il Progetto di programmazione 15 del Capitolo 8 in modo che il programma chieda all'utente di immettere il nome di un file contenente il messaggio che deve essere cifrato:

Enter name of file to be encrypted: message.txt

Enter shift amount (1-25): 3

Il programma deve scrivere il messaggio cifrato in un file con lo stesso nome di quello originale, ma con l'aggiunta dell'estensione `.enc`. In questo esempio, il nome del file originale è `message.txt`, di conseguenza il messaggio cifrato verrà inserito in un file chiamato `message.txt.enc`. Non c'è limite alla dimensione del file che deve essere cifrato o sulla lunghezza di ogni riga del file.

15. Modificate il programma `justify` della Sezione 15.3 in modo che legga da un file di testo e scriva all'interno di un altro. Fate in modo che il programma ottenga il nome di entrambi i file dalla riga di comando.
16. Modificate il programma `fcopy.c` della Sezione 22.4 in modo che utilizzi le funzioni `fread` e `fwrite` per copiare il file in blocchi di 512 byte (naturalmente l'ultimo blocco può contenere meno di 512 byte).
17. Scrivete un programma che legga da un file una serie di numeri telefonici e li visualizzi in un formato standard. Ogni riga del file dovrà contenere un singolo numero di telefono, ma i numeri potranno essere scritti in diversi formati. Potete assumere che ogni riga contenga 10 cifre, che possono essere mischiate con altri caratteri (che devono essere ignorati). Per esempio, supponete che il file contenga le seguenti righe:

```
404.817.6900
(215) 686-1776
312-746-6000
877 275 5273
6173434200
```

L'output del programma deve avere il seguente aspetto:

```
(404) 817-6900
(215) 686-1776
(312) 746-6000
(877) 275-5273
(617) 343-4200
```

Fate in modo che il programma ottenga il nome del file dalla riga di comando.

18. Scrivete un programma che legga degli interi da un file di testo il cui nome viene passato come argomento della riga di comando. Ogni riga può contenere un numero qualsiasi di interi (anche nessuno) separati da uno o più spazi. Il programma dovrà visualizzare il più grande tra i numeri contenuti nel file, quello più piccolo e quello mediano (il numero più vicino alla metà se gli interi fossero ordinati). Se il file contiene un numero pari di interi, nel mezzo ci sono due numeri. In tal caso il programma dovrà visualizzare la loro media (arrotondata per difetto). Potete assumere che il file contenga non più di 10.000 interi. Suggerimento: salvate gli interi in un vettore e poi ordinatevi.
19. (a) Scrivete un programma che converta un file di testo Windows in un file di testo UNIX (si veda la Sezione 22.1 per una discussione delle differenze tra i file di testo dei due sistemi).
- (b) Scrivete un programma che converta un file di testo UNIX in un file di testo Windows.

Per ognuno dei due punti fate in modo che il programma ottenga i nomi di entrambi i file dalla riga di comando. Suggerimento: aprirete il file di input in modalità "rb" e il file di output in modalità "wb".

23 Supporto per numeri e caratteri

Questo capitolo descrive i cinque più importanti header della libreria che forniscono supporto per lavorare con i numeri, i caratteri e le stringhe di caratteri. Le Sezioni 23.1 e 23.2 trattano gli header `<float.h>` e `<limits.h>`, i quali contengono macro che descrivono le caratteristiche dei tipi numerici e di quelli carattere. Le Sezioni 23.3 e 23.4 descrivono l'header `<math.h>`, che fornisce delle funzioni matematiche. La Sezione 23.3 tratta la versione C89 di questo header mentre la Sezione 23.4 descrive le numerose aggiunte dello standard C99, che sono state trattate separatamente. Le Sezioni 23.5 e 23.6 sono dedicate agli header `<ctype.h>` e `<string.h>`, i quali forniscono delle funzioni rispettivamente per i caratteri e le stringhe di caratteri.

Il C99 introduce diversi nuovi header che gestiscono i numeri, i caratteri e le stringhe. Gli header `<wchar.h>` e `<wctype.h>` vengono approfonditi nel Capitolo 25. Il Capitolo 27 invece illustra gli header `<complex.h>`, `<fenv.h>`, `<inttypes.h>`, `<stdin.h>` e `<tgmath>`.

23.1 L'header `<float.h>`: caratteristiche dei tipi a virgola mobile

L'header `<float.h>` fornisce delle macro che definiscono il range e l'accuratezza dei tipi `float`, `double` e `long double`. In questo header non ci sono né tipi né funzioni.

Due macro si applicano a tutti i tipi `float`. La prima delle due, la `FLT_ROUNDS`, rappresenta la direzione corrente per l'arrotondamento nelle addizioni a virgola mobile [**direzione di arrotondamento > 23.4**]. La Tabella 23.1 illustra tutti i possibili valori di `FLT_ROUNDS` (i valori non elencati nella tabella indicano dei comportamenti dipendenti dall'implementazione).

Tabella 23.1 Direzioni di arrotondamento

Valore	Significato
-1	Indeterminabile
0	Verso lo zero
1	Verso il più vicino
2	Verso l'infinito positivo
3	Verso l'infinito negativo

A differenza delle altre macro definite in `<float.h>`, che rappresentano delle espressioni costanti, il valore della `FLT_ROUNDS` può cambiare durante l'esecuzione (la funzione `fesetround` [funzione `fesetround > 27.6`] permette di modificare la direzione di arrotondamento corrente). La seconda macro, la `FLT_RADIX`, specifica la radice della rappresentazione esponenziale. Il suo valore minimo è pari a 2 (indicante la rappresentazione binaria).

Le altre macro dell'header, che verranno rappresentate in una serie di tabelle, descrivono le caratteristiche di tipi specifici. Ogni macro inizia con `FLT`, `DBL` o `LDBL`, a seconda che si riferiscono al tipo `float`, `double` o `long double`. Lo standard C fornisce delle definizioni estremamente dettagliate per queste macro, la nostra descrizione sarà meno precisa ma di più facile comprensione. Per alcune delle macro, le tabelle indicano il valore massimo e quello minimo dettati dallo standard.

La Tabella 23.2 elenca le macro che definiscono il numero di cifre significative garantite per ogni tipo a virgola mobile.

Tabella 23.2 Macro per le cifre significative presenti in `<float.h>`

Nome	Valore	Descrizione
<code>FLT_MANT_DIG</code>		Numero di cifre significative (base <code>FLT_RADIX</code>)
<code>DBL_MANT_DIG</code>		
<code>LDBL_MANT_DIG</code>		
<code>FLT_DIG</code>	≥ 6	Numero di cifre significative (base 10)
<code>DBL_DIG</code>	≥ 10	
<code>LDBL_DIG</code>	≥ 10	

La Tabella 23.3 elenca le macro che hanno a che fare con gli esponenti.

Tabella 23.3 Macro per gli esponenti presenti in `<float.h>`

Nome	Valore	Descrizione
<code>FLT_MIN_EXP</code>		La più piccola (più negativa) potenza alla quale <code>FLT_RADIX</code> può essere elevata
<code>DBL_MIN_EXP</code>		
<code>LDBL_MIN_EXP</code>		
<code>FLT_MIN_10_EXP</code>	≤ -37	La più piccola (più negativa) potenza alla quale il numero 10 può essere elevato
<code>DBL_MIN_10_EXP</code>	≤ -37	
<code>LDBL_MIN_10_EXP</code>	≤ -37	
<code>FLT_MAX_EXP</code>		La potenza più grande alla quale <code>FLT_RADIX</code> può essere elevata
<code>DBL_MAX_EXP</code>		
<code>LDBL_MAX_EXP</code>		
<code>FLT_MAX_10_EXP</code>	$\geq +37$	La potenza più grande alla quale il numero 10 può essere elevato
<code>DBL_MAX_10_EXP</code>	$\geq +37$	
<code>LDBL_MAX_10_EXP</code>	$\geq +37$	

La Tabella 23.4 elenca le macro che descrivono quanto possono essere grandi i numeri, quanto possono avvicinarsi allo zero e quanto possono essere vicini due numeri consecutivi.

Tabella 23.4 Le macro max, min ed epsilon di `<float.h>`

Nome	Valore	Descrizione
FLT_MAX	$\geq 10^{37}$	Il più grande valore finito
DBL_MAX	$\geq 10^{37}$	
LDBL_MAX	$\geq 10^{37}$	
FLT_MIN	$\leq 10^{-37}$	Il più piccolo numero positivo
DBL_MIN	$\leq 10^{-37}$	
LDBL_MIN	$\leq 10^{-37}$	
FLT_EPSILON	$\leq 10^{-5}$	La più piccola differenza tra due numeri che sia rappresentabile
DBL_EPSILON	$\leq 10^{-9}$	
LDBL_EPSILON	$\leq 10^{-9}$	

C99

Il C99 fornisce altre due macro: DECIMAL_DIG e FLT_EVAL_METHOD. DECIMAL_DIG rappresenta il numero di cifre significative (in base 10) presenti nel più grande tipo a virgola mobile supportato. Il suo valore minimo è pari a 10. Il valore di FLT_EVAL_METHOD indica se un'implementazione esegue dell'aritmetica a virgola mobile usando un intervallo di valori e una precisione più grandi di quanto strettamente richiesto. Se questa macro ha valore 0, per esempio, allora la somma di due valori float verrebbe eseguita nel modo normale. Se la macro ha valore 1, invece, i valori float vengono convertiti in double prima che la somma venga eseguita. La Tabella 23.5 elenca i possibili valori di FLT_EVAL_METHOD (valori negativi non presenti nella tabella indicano dei comportamenti definiti dall'implementazione).

Tabella 23.5 Metodi di calcolo

Valore	Significato
-1	Indeterminabile
0	Valuta tutte le operazioni e le costanti esattamente nell'intervallo e con la precisione del tipo
1	Valuta tutte le operazioni e le costanti di tipo float e double nell'intervallo di valori e con la precisione del tipo double
2	Valuta tutte le operazioni e le costanti nell'intervallo di valori e con la precisione del tipo long double

La maggior parte delle macro presenti in `<float.h>` sono di interesse solo per gli esperti in analisi numerica, il che lo rende probabilmente l'header meno usato della libreria standard.

23.2 L'header <limits.h>: dimensioni dei tipi interi

L'header <limits.h> fornisce macro che definiscono l'intervallo di valori per tutti i tipi interi (inclusi i tipi carattere). Questo header non dichiara né tipi né funzioni.

Un insieme di macro presenti in <limits.h> ha a che fare con i tipi carattere: char, signed char e unsigned char. La Tabella 23.6 elenca queste macro e illustra i valori massimi e minimi assumibili da ognuna di queste.

Le altre macro dell'header trattano i rimanenti tipi interi: short int, unsigned short int, int, unsigned int, long int e unsigned long int. La Tabella 23.7 elenca queste macro e mostra i valori massimi e minimi di ognuna. Viene data anche la formula usata per calcolare i diversi valori. Notate che il C99 fornisce tre macro che descrivono le caratteristiche dei tipi long long int.

C99

Tabella 23.6 Macro per i caratteri presenti in <limits.h>

Nome	Valore	Descrizione
CHAR_BIT	≥ 8	Numero di bit per byte
SCHAR_MIN	≤ -127	Minimo valore signed char
SCHAR_MAX	$\geq +127$	Massimo valore signed char
UCHAR_MAX	≥ 255	Massimo valore unsigned char
CHAR_MIN	†	Minimo valore char
CHAR_MAX	††	Massimo valore char
MB_LEN_MAX	≥ 1	Massimo numero di byte presenti in un carattere multibyte in tutte le localizzazioni supportate (vedi Sezione 25.2)

[†]CHAR_MIN è uguale a SCHAR_MIN se char viene trattato come un tipo con segno, altrimenti è uguale a zero.

^{††}CHAR_MAX è dello stesso valore di SCHAR_MAX o UCHAR_MAX a seconda che char venga trattato come un tipo con o senza segno.

Tabella 23.7 Macro per i caratteri presenti in <limits.h>

Nome	Valore	Formula	Descrizione
SHRT_MIN	≤ 32767	$-(2^{15}-1)$	Minimo valore short int
SHRT_MAX	$\geq +32767$	$2^{15}-1$	Massimo valore short int
USHRT_MAX	≥ 65535	$2^{16}-1$	Massimo valore unsigned short int
INT_MIN	≤ -32767	$-(2^{15}-1)$	Minimo valore int
INT_MAX	$\geq +32767$	$2^{15}-1$	Massimo valore int
UINT_MAX	≥ 65535	$2^{16}-1$	Massimo valore unsigned int
LONG_MIN	≤ -2147483647	$-(2^{31}-1)$	Minimo valore long int
LONG_MAX	$\geq +2147483647$	$2^{31}-1$	Massimo valore long int
ULONG_MAX	≥ 4294967295	$2^{32}-1$	Massimo valore unsigned long int

Nome	Valore	Formula	Descrizione
LLONG_MIN ^t	≤ 9223372036854775807	$-(2^{63}-1)$	Minimo valore long long int
LLONG_MAX ^t	$\geq +9223372036854775807$	$2^{63}-1$	Massimo valore long long int
ULLONG_MAX ^t	$\geq 18446744073709551615$	$2^{64}-1$	Massimo valore unsigned long long int

^tsolo C99

Le macro presenti in <limits.h> sono comode per controllare se un compilatore supporta una particolare dimensione. Per esempio, per determinare se il tipo int è in grado di contenere valori grandi quanto il numero 100.000, possiamo usare le seguenti direttive di preprocessamento:

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

Se il tipo int non è adeguato, la direttiva #error [direttiva #error > 14.5] fa sì che il preprocessore visualizzi un messaggio di errore.

Facendo un passo ulteriore, possiamo usare le macro presenti in <limits.h> per aiutare un programma a scegliere come rappresentare un tipo. Diciamo che le variabili di tipo Quantity debbano contenere interi grandi quanto 100.000. Se INT_MAX è maggiore di 100.000 possiamo definire il tipo Quantity come int, altrimenti dovremo definirlo come long int:

```
#if INT_MAX >= 100000 .
typedef int Quantity;
#else
typedef long int Quantity;
#endif
```

23.3 L'header <math.h> (C89): matematica

Le funzioni della versione C89 dell'header <math.h> ricadono dentro cinque gruppi:

- funzioni trigonometriche;
- funzioni iperboliche;
- funzioni esponenziali e logaritmiche;
- funzioni di elevamento a potenza;
- funzioni di intero più vicino, valore assoluto e resto.

Il C99 aggiunge a questo gruppo un buon numero di funzioni, oltre a introdurre altre categorie di funzioni matematiche. Le modifiche del C99 all'header <math.h> sono così estese che le tratteremo in una sezione separata. In questo modo i lettori che sono interessati principalmente alla versione C89 dell'header (o che utilizzano un compilatore che non supporta il C99) non saranno sopraffatti da tutte le aggiunte del C99.

Prima di scavare all'interno di <math.h> e delle sue funzioni, guardiamo brevemente come queste ultime gestiscono gli errori.

Errori

Le funzioni presenti in `<math.h>` gestiscono gli errori in modo diverso da quello delle altre funzioni di libreria. Quando si verifica un errore, la maggior parte delle funzioni di `<math.h>` salvano un codice di errore in una speciale variabile chiamata `errno` (dichiarata nell'header `<errno.h>` [header `<errno.h>` > 24.2]). In aggiunta, quando il valore restituito da una funzione è maggiore del più grande valore `double`, le funzioni presenti in `<math.h>` restituiscono uno speciale valore rappresentato dalla macro `HUGE_VAL` (definita in `<math.h>`). La macro `HUGE_VAL` è di tipo `double` ma non è necessariamente un numero normale (lo standard IEEE per l'aritmetica in virgola mobile definisce un valore chiamato "infinity" [infinity > 23.4], una scelta logica per la macro `HUGE_VAL`).

Le funzioni contenute in `<math.h>` rilevano due tipi di errori.

- **Errori di dominio:** un argomento è al di fuori del dominio di una funzione. Se si verifica un errore di dominio, il valore restituito dalla funzione è definito dall'implementazione mentre in `errno` viene salvato il valore `EDOM` (*domain error*). In alcune implementazioni di `<math.h>`, quando si verifica un errore di dominio, le funzioni restituiscono uno speciale valore conosciuto come `NaN` (*not a number*) [NaN > 23.4].
- **Errore di intervallo:** il valore restituito da una funzione è al di fuori dell'intervallo dei valori `double`. Se una funzione restituisce un valore il cui valore assoluto è troppo grande (*overflow*), questa restituirà il valore `HUGE_VAL` con segno positivo o negativo a seconda del segno posseduto dal risultato corretto. In aggiunta in `errno` viene salvato il valore `ERANGE` (*range error*). Se il valore assoluto del valore restituito è troppo piccolo per essere rappresentabile (*underflow* [*underflow* > 23.4]), la funzione restituisce uno zero. Alcune implementazioni inoltre salvano in `errno` il valore `ERANGE`.

In questa sezione ignoreremo la possibilità di errori per il resto. Tuttavia la descrizione delle funzioni presentata nell'**Appendice D** spiega le circostanze che conducono a ogni tipo di errore.

Funzioni trigonometriche

```
double acos(double x);
double asin(double x);
double atan(double x);
double atan2(double y, double x);
double cos(double x);
double sin(double x);
double tan(double x);
```

`cos`
`sin`
`tan` Le funzioni `cos`, `sin` e `tan` calcolano rispettivamente il coseno, il seno e la tangente. Se la macro `PI` è definita come 3.14159265, passare `PI/4` a queste funzioni produce i seguenti risultati:

$$\cos(\text{PI}/4) \Rightarrow 0.707107$$

$\sin(\text{PI}/4) \Rightarrow 0.707107$

$\tan(\text{PI}/4) \Rightarrow 1.0$

Prestate attenzione al fatto che gli argomenti delle funzioni cos, sin e tan vengono espressi in radianti e non in gradi.

acos Le funzioni acos, asin e atan calcolano l'arcocoseno, l'arcoseno e l'arcotangente:

asin

atan $\cdot \quad \text{acos}(1.0) \Rightarrow 0.0$

$\text{asin}(1.0) \Rightarrow 1.5708$

$\text{atan}(1.0) \Rightarrow 0.785398$

Applicare la funzione acos al valore restituito dalla funzione cos non ci fa ottenere necessariamente l'argomento originale di quest'ultimo. Il motivo è che la acos restituisce sempre un valore compreso tra 0 e π , mentre asin e atan restituiscono sempre un valore compreso tra $-\pi/2$ e $\pi/2$.

atan2 La funzione atan2 calcola l'arcotangente di y/x dove y è il primo argomento della funzione e x è il secondo. Il valore restituito dalla atan2 è compreso tra $-\pi$ e π . La chiamata atan(x) è equivalente alla chiamata atan2(x , 1.0).

Funzioni iperboliche

```
double cosh(double x);
double sinh(double x);
double tanh(double x);
```

cosh Le funzioni cosh, sinh e tanh calcolano rispettivamente il seno iperbolico, il seno iperbolico e la tangente iperbolica:

sinh

tanh

$\cosh(0.5) \Rightarrow 1.12763$

$\sinh(0.5) \Rightarrow 0.521095$

$\tanh(0.5) \Rightarrow 0.462117$

Gli argomenti di cosh, sinh e tanh devono essere espressi in radianti e non in gradi.

Funzioni esponenziali e logaritmiche

```
double exp(double x);
double frexp(double value, int *exp);
double ldexp(double x, int exp);
double log(double x);
double log10(double x);
double modf(double value, double *iptr);
```

exp La funzione exp restituisce il valore di e elevato alla potenza rappresentata dall'argomento:

$\exp(3.0) \Rightarrow 20.0855$

log

log10

La funzione log è l'inversa della funzione exp (calcola il logaritmo in base e di un numero). La funzione log10 calcola il logaritmo in base 10:

```
log(20.0855) ⇒ 3.0
log10(1000) ⇒ 3.0
```

Calcolare il logaritmo in una base diversa da e o 10 non è difficile. La seguente funzione, per esempio, calcola il logaritmo di x in base b , con x e b arbitrari:

```
double log_base(double x, double b)
{
    return log(x) / log(b);
}
```

modf Le funzioni `modf` e `frexp` scompongono un valore `double` in due parti. La `modf` divide suo primo argomento nella sua parte intera e in quella frazionaria. Restituisce la parte frazionaria e salva quella intera nell'oggetto puntato dal secondo argomento:

```
modf(3.14159, &int_part) ⇒ 0.14159 (a int_part viene assegnato il valore 3.0)
```

Sebbene `int_part` debba essere di tipo `double`, possiamo sempre applicare successivamente `cast` verso i tipi `int` o `long int`.

frexp La funzione `frexp` divide un numero a virgola mobile in una mantissa f e in un esponente n in modo che il numero originale sia uguale a $f \times 2^n$, dove sia ha che $0.5 \leq f < 1$ oppure $f = 0$. La funzione restituisce f e salva n nell'oggetto (intero) puntato dal secondo argomento:

```
frexp(12.0, &exp) ⇒ .75 (a exp viene assegnato il valore 4)
```

```
frexp(0.25, &exp) ⇒ 0.5 (a exp viene assegnato il valore -1)
```

ldexp La funzione `ldexp` annulla quanto fatto da `frexp` combinando una mantissa e un esponente in un singolo numero:

```
ldexp(.75, 4) ⇒ 12.0
```

```
ldexp(0.5, -1) ⇒ 0.25
```

In generale, la chiamata `ldexp(x, exp)` restituisce $x \times 2^{\text{exp}}$.

Le funzioni `modf`, `frexp` e `ldexp` vengono utilizzate principalmente da altre funzioni presenti in `<math.h>` e raramente vengono chiamate direttamente dai programmi.

Funzioni di elevamento a potenza

<pre>double pow(double x, double y);</pre> <pre>double sqrt(double x);</pre>
--

pow La funzione `pow` eleva il suo primo argomento alla potenza specificata dal secondo argomento:

```
pow(3.0, 2.0) ⇒ 9.0
```

```
pow(3.0, 0.5) ⇒ 1.73205
```

```
pow(3.0, -3.0) ⇒ 0.037037
```

sqrt La funzione `sqrt` calcola la radice quadrata:

```
sqrt(3.0) ⇒ 1.73205
```

Per calcolare la radice quadrata è preferibile usare la `sqrt` rispetto alla `pow` perché solitamente la prima è una funzione molto più veloce.

Funzioni di intero più vicino, valore assoluto e resto

```
double ceil(double x);
double fabs(double x);
double floor(double x);
double fmod(double x, double y);
```

`ceil`
`floor`

La funzione `ceil` (*ceiling*) restituisce un valore `double` che rappresenta il più piccolo intero che è maggiore o uguale al suo argomento. La funzione `floor` restituisce il più grande intero che è minore o uguale al suo argomento:

```
ceil(7.1)    => 8.0
ceil(7.9)    => 8.0
ceil(-7.1)   => -7.0
ceil(-7.9)   => -7.0

floor(7.1)   => 7.0
floor(7.9)   => 7.0
floor(-7.1)  => -8.0
floor(-7.9)  => -8.0
```

In altre parole, la `ceil` arrotonda per eccesso all'intero più vicino, mentre la `floor` arrotonda per difetto. Il C89 non è provvisto di una funzione standard che arrotondi all'intero più vicino, tuttavia possiamo crearne una nostra utilizzando `ceil` e `floor`:

```
double round_nearest(double x)
{
    return x < 0.0 ? ceil(x - 0.5) : floor(x + 0.5);
}
```

C99

Come vedremo nella prossima sezione, il C99 fornisce diverse funzioni che arrotondano all'intero più vicino.

`fabs`

La funzione `fabs` calcola il valore assoluto di un numero:

```
fabs(7.1)    => 7.1
fabs(-7.1)   => 7.1
```

`fmod`

La funzione `fmod` restituisce il resto ottenuto quando il primo argomento viene diviso per il secondo:

```
fmod(5.5, 2.2) => 1.1
```

Il C non ammette che l'operatore `%` abbia operandi a virgola mobile, tuttavia la funzione `fmod` è un più che valido sostituto.

23.4 L'header `<math.h>` (C99): matematica

La versione C99 dell'header `<math.h>` include l'intera versione C89 oltre a un sacco di funzioni, macro e tipi aggiuntivi. Le modifiche a questo header sono così numerose da meritare di essere trattate separatamente. Ci sono diverse ragioni per le quali il comitato dello standard ha aggiunto così tante funzionalità all'header `<math.h>`.

- **Fornire un supporto migliore allo standard floating point IEEE.** Il C99 non obbliga all'uso dello standard IEEE, è possibile usare anche altri modi per rappresentare i numeri a virgola mobile. Tuttavia bisogna dire che la stragrande maggioranza dei programmi vengono eseguiti su sistemi che supportano questo standard.
- **Fornire un maggiore controllo sull'aritmetica a virgola mobile.** Un migliore controllo nell'aritmetica a virgola mobile può permettere ai programmi di ottenere una maggiore accuratezza e velocità.
- **Rendere il C più attraente per i programmatori Fortran.** L'intenzione dietro l'aggiunta di molte funzioni matematiche e di altri miglioramenti che sono stati introdotti nel C99 (come il supporto per i numeri complessi) era quella di rendere il C più attrattivo per i programmatori che in passato avevano utilizzato altri linguaggi di programmazione (principalmente il Fortran).

Un'altra ragione per aver trattato separatamente l'header `<math.h>` del C99 è costituita dal fatto che tutti questi argomenti sono di scarso interesse per il programmatore C medio. Quelli che utilizzano il C per le sue applicazioni tradizionali che includono la programmazione di sistema e dei sistemi embedded, probabilmente non avranno bisogno delle nuove funzioni aggiunte dal C99. Tuttavia i programmatori che sviluppano applicazioni ingegneristiche, scientifiche o matematiche potrebbero trovare queste funzioni piuttosto utili.

Lo standard floating point dell'IEEE

Una motivazione per le modifiche all'header `<math.h>` è un miglior supporto dello standard IEEE 754, ovvero la rappresentazione dei numeri a virgola mobile più utilizzata. Il titolo completo dello standard è: *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 754-1985)*. È conosciuto anche come IEC 60559, che è il modo nel quale vi si riferisce lo standard C99.

La Sezione 7.2 ha descritto alcune proprietà basilari dello standard IEEE. Abbiamo visto che lo standard fornisce due formati principali per i numeri a virgola mobile: la precisione singola (32 bit) e la precisione doppia (64 bit). I numeri vengono memorizzati secondo la notazione scientifica, dove ogni numero è costituito da tre parti: un segno, un esponente e una mantissa. Una conoscenza così limitata dello standard IEEE è sufficiente per usare la versione C89 dell'header `<math.h>`. Capire la versione C99, invece, richiede una conoscenza più approfondita. Ecco alcune informazioni aggiuntive di cui abbiamo bisogno:

- **Zero positivo/negativo.** Uno dei bit del formato IEEE per i numeri a virgola mobile rappresenta il segno del numero. Ne consegue che il numero zero possa essere sia positivo che negativo a seconda del valore di questo bit. Il fatto che lo zero abbia due rappresentazioni può richiederci, a volte, di trattarlo diversamente dagli altri numeri.

- **Numeri subnormali.** Quando viene eseguita una funzione in virgola mobile, il risultato può essere troppo piccolo per essere rappresentato, una condizione conosciuta come **underflow**. Pensate a cosa accadrebbe nel caso divideste ripetutamente un numero usando una calcolatrice tascabile: il risultato può essere pari a zero a causa del fatto che questo diventa troppo piccolo per essere rappresentabile con il formato numerico usato dalla calcolatrice. Lo standard IEEE ha un modo per ridurre l'impatto di questo fenomeno. I normali numeri a virgola mobile vengono memorizzati in un formato "normalizzato" nel quale il numero viene scalato in modo che ci sia esattamente una cifra a sinistra del punto binario. Quando il numero diventa sufficientemente piccolo però, viene memorizzato in un formato diverso che non è normalizzato. Questi **numeri subnormali** (conosciuti anche come **numeri denormalizzati** o **denormal**) possono essere più piccoli dei numeri normalizzati. Il trade-off è che questi diventano meno accurati man mano che diventano più piccoli.
- **Valori speciali.** Tutti i formati a virgola mobile permettono la rappresentazione di tre valori speciali: **infinito positivo**, **infinito negativo** e **NaN (not a number)**. La divisione di un numero positivo per zero genera l'infinito positivo. La divisione di un numero negativo per zero dà un infinito negativo. Il risultato di un'operazione matematicamente indefinita, come dividere lo zero per zero, corrisponde a NaN. È più corretto dire "il risultato è *un* NaN" invece di "il risultato è NaN" perché lo standard IEEE possiede rappresentazioni multiple per NaN. L'esponente di un valore NaN ha tutti i bit a 1, ma la mantissa può essere costituita da una qualsiasi sequenza di bit diversi da zero. I valori speciali possono essere degli operandi per le operazioni successive. L'infinito si comporta esattamente come nella matematica ordinaria. Per esempio: dividendo un numero positivo per l'infinito positivo si ottiene uno zero (osservate che un'espressione aritmetica può produrre un infinito come risultato intermedio ma avere un valore complessivo diverso da infinito). Eseguire una qualsiasi operazione su NaN restituisce NaN come risultato.
- **Direzione di arrotondamento.** Quando un numero non può essere memorizzato in modo esatto usando la rappresentazione a virgola mobile, la **direzione di arrotondamento** (o **modo di arrotondamento**) corrente determina quale numero a virgola mobile verrà scelto per rappresentare il numero stesso. Ci sono quattro direzioni di arrotondamento: (1) *Arrotondamento verso il più vicino*. Arrotonda verso il valore rappresentabile più vicino. Se un numero cade a metà strada tra due valori, viene arrotondato al valore "pari" (quello il cui bit meno significativo è uguale a zero). (2) *Arrotondamento verso lo zero*. (3) *Arrotondamento verso l'infinito positivo*. (4) *Arrotondamento verso l'infinito negativo*. La direzione di arrotondamento di default è quella verso il numero più vicino.
- **Eccezioni.** Ci sono cinque tipi di eccezioni floating point: *overflow*, *underflow*, *divisione per zero*, *operazione non valida* (il risultato di un'operazione aritmetica era NaN) e *operazione inesatta* (il risultato di un'operazione numerica doveva essere arrotondato). Quando una di queste condizioni viene rilevata, diciamo che è stata sollevata un'eccezione.

Tipi

Il C99 aggiunge due tipi a `<math.h>`: `float_t` e `double_t`. Il tipo `float_t` è “grande” almeno quanto il tipo `float` (in che significa che può corrispondere al tipo `float` o a qualsiasi tipo più grande, come il `double`). Analogamente, `double_t` deve essere grande almeno quanto il tipo `double` (e deve essere grande almeno quanto `float_t`). Questi tipi vengono forniti al programmatore che sta cercando di massimizzare la performance dell’aritmetica a virgola mobile. Il tipo `float_t` dovrebbe essere il più efficiente tipo a virgola mobile che è grande almeno quanto il tipo `float`, mentre `double_t` dovrebbe essere il più efficiente tipo a virgola mobile che è grande almeno quanto il tipo `double`.

Così come si può vedere nella Tabella 23.8, i tipi `float_t` e `double_t` sono legati alla macro `FLT_EVAL_METHOD`.

Tabella 23.8 Relazione esistente tra `FLT_EVAL_METHOD` e i tipi `float_t` e `double_t`

Valore di <code>FLT_EVAL_METHOD</code>	Significato di <code>float_t</code>	Significato di <code>double_t</code>
0	<code>float</code>	<code>double</code>
1	<code>double</code>	<code>double</code>
2	<code>long double</code>	<code>long double</code>
Altri	definito dall’implementazione	definito dall’implementazione

Macro

Il C99 aggiunge diverse macro all’header `<math.h>`, ma qui ne menzioneremo solamente due. La macro `INFINITY` rappresenta la versione `float` dell’infinito positivo senza segno (se l’implementazione non supporta l’infinito, allora la macro rappresenta un valore `float` che va in overflow al momento della compilazione). La macro `NAN` rappresenta la versione `float` di *not a number*. Più precisamente rappresenta una versione “tranquilla” di `NaN` (ovvero una che non solleva un’eccezione se viene usata in un’espressione aritmetica). Se i `NaN` “tranquilli” non sono supportati, la macro `NAN` non viene definita.

Tratteremo le macro parametriche che sono presenti in `<math.h>` in una sezione più avanti nel testo, assieme alle funzioni normali. Le macro che sono rilevanti solamente per una funzione verranno descritte assieme alla funzione stessa.

Errori

Per la gran parte la versione C99 di `<math.h>` gestisce gli errori nello stesso modo della versione C89. Tuttavia, ci sono un paio di questioni che dobbiamo discutere.

Per prima cosa il C99 fornisce diverse macro che danno alle implementazioni scelta di come segnalare gli errori: attraverso un valore salvato in `errno`, con un’eccezione floating point o con entrambi i modi precedenti. Le macro `MATH_ERRNO` e `MATH_ERREXCEPT` rappresentano rispettivamente le costanti intere 1 e 2. Una terza ma-

cro, `math_errhandling`, rappresenta un'espressione `int` il cui valore è pari a `MATH_ERRNO`, `MATH_ERREXCEPT` o l'`or` bitwise dei due valori (è anche possibile che `math_errhandling` non sia realmente una macro, potrebbe essere un identificatore con collegamento esterno). Il valore di `math_errhandling` non può essere modificato all'interno di un programma.

Guardiamo ora cosa succede quando si verifica un errore di dominio durante una chiamata a una delle funzioni di `<math.h>`. Lo standard C89 dice che il valore `EDOM` viene salvato all'interno della variabile `errno`. Lo standard C99, da parte sua, dice che il valore `EDOM` viene salvato in `errno` se l'espressione `math_errhandling & MATH_ERRNO` è diversa da zero (ovvero se il bit `MATH_ERRNO` è pari a 1). Se l'espressione `math_errhandling & MATH_ERREXCEPT` è diversa da zero viene sollevata l'eccezione floating point `invalid`. Quindi entrambe le azioni sono possibili a seconda del valore di `math_errhandling`.

Guardiamo infine alle azioni che si verificano quando viene rilevato un errore di intervallo durante una chiamata a funzione. Ci sono due casi che dipendono dalla grandezza del valore restituito dalla funzione.

Overflow. Se il valore assoluto è troppo grande, lo standard C89 richiede che la funzione restituiscia il valore `HUGE_VAL` positivo o negativo a seconda del segno del risultato corretto. Inoltre dentro `errno` viene salvato il valore `ERANGE`. Lo standard C99 descrive un insieme più complicato di azioni quando si verifica un overflow:

- Se è attiva la modalità di arrotondamento di default o il valore restituito è un “infinito esatto” (come `log(0.0)`), allora la funzione restituisce `HUGE_VAL`, `HUGE_VALF` o `HUGE_VALL` a seconda del tipo restituito (`HUGE_VALF` e `HUGE_VALL` sono nuovi del C99 e rappresentano rispettivamente le versioni `float` e `long double` di `HUGE_VAL`. Così come `HUGE_VAL`, anche questi possono rappresentare l’infinito positivo). Il valore restituito avrà il segno del risultato corretto.
- Se il valore di `math_errhandling & MATH_ERRNO` è diverso da zero, il valore `ERANGE` viene salvato in `errno`.
- Se il valore di `math_errhandling & MATH_ERREXCEPT`, nel caso il risultato sia un infinito esatto si verifica l’eccezione floating point chiamata *divide-by-zero*. Altrimenti viene sollevata l’eccezione *overflow*.

Underflow. Se il valore assoluto è troppo piccolo per essere rappresentato, lo standard C89 richiede che la funzione restituiscia uno zero. Alcune implementazioni possono anche salvare il valore `ERANGE` all'interno di `errno`. Lo standard C99 prescrive un insieme diverso di azioni:

- La funzione restituisce un valore il cui valore assoluto è minore o uguale a quello del più piccolo numero positivo normalizzato appartenente al tipo restituito dalla funzione (questo valore dovrebbe essere zero o un numero subnormale).
- Se il valore di `math_errhandling & MATH_ERRNO` è diverso da zero, un’implementazione può salvare `ERANGE` all'interno di `errno`.
- Se il valore di `math_errhandling & MATH_ERREXCEPT` è diverso da zero, un’implementazione può sollevare l’eccezione floating point *underflow*.

Notate la parola “può” presente negli ultimi due casi. Per ragioni di efficienza, un’implementazione non è obbligata a modificare `errno` o a sollevare l’eccezione *underflow*.

Funzioni

Ora siamo pronti per trattare le funzioni che sono state aggiunte a `<math.h>` dal C99. Tratteremo le funzioni in gruppi, usando le stesse categorie utilizzate dallo standard. Queste categorie differiscono in qualche modo dalla Sezione 23.3 che derivava dallo standard C89.

Una delle modifiche più grandi nella versione C99 di `<math.h>` consiste nell'aggiunta di due versioni aggiuntive per la maggior parte delle funzioni. Nel C89, c'è una singola versione per ogni funzione matematica, la quale, tipicamente, accetta almeno un argomento di tipo `double` e/o restituisce un valore `double`. Nel C99, invece, ci sono due versioni aggiuntive: una per il tipo `float` e una per il tipo `long double`. I nomi di queste funzioni sono identici a quello originale ad eccezione del fatto che differiscono da questo per il suffisso `f` o `l`. Per esempio, la funzione originale `sqrt`, che effettua la radice quadrata di un valore `double`, ora è accompagnata dalla `sqrtf` (la versione `float`) e dalla `sqrtd` (la versione `long double`). Elencheremo tutti i prototipi per le nuove versioni (in corsivo, così come abbiamo fatto per tutte le funzioni che sono nuove del C99), ma non le descriveremo ulteriormente visto che sono praticamente identiche alle loro controparti C89.

La versione C99 di `<math.h>` include anche diverse funzioni completamente nuove (oltre che macro parametriche). Daremo una breve descrizione di ognuna. Come nella Sezione 23.3 non discuteremo delle condizioni di errore di queste funzioni, tuttavia queste informazioni sono fornite dall'Appendice D, che elenca tutte le funzioni della libreria standard in ordine alfabetico. Non elencheremo i nomi di tutte le nuove funzioni nel margine sinistro, ma verrà mostrato solo il nome della funzione principale. Per esempio: ci sono tre nuove funzioni che calcolano l'arcocoseno iperbolico: `acosh`, `acoshf` e `acoshl`. Verrà descritta la funzione `acosh` e nel margine sinistro verrà visualizzato solo il suo nome.

Tenete in mente che molte delle nuove funzioni sono altamente specializzate. Di conseguenza la descrizioni di queste funzioni può sembrare un po' sommaria. Una discussione riguardo a cosa servano queste funzioni è fuori dagli scopi di questo libro.

Macro di classificazione

```
int fpclassify (real-floating x);
int isfinite (real-floating x);
int isinf (real-floating x),
int isnan (real-floating x),
int isnormal (real-floating x),
int signbit (real-floating x);
```

La nostra prima categoria consiste di alcune macro parametriche che vengono utilizzate per determinare se valore a virgola mobile è un numero "normale" o se è un valore speciale come infinito o NaN. Le macro presenti in questo gruppo sono progettate per accettare argomenti di qualsiasi tipo reale a virgola mobile (`float`, `double` o `long double`).

fpclassify La macro *fpclassify* classifica il suo argomento restituendo il valore di una delle macro di classificazione presenti in Tabella 23.9. Un'implementazione può supportare altre classificazioni definendo delle macro aggiuntive i cui nomi inizino per *FP_* e una lettera maiuscola.

Tabella 23.9 Macro di classificazione per i numeri

Nome	Significato
<i>FP_INFINITE</i>	Infinito (positivo o negativo)
<i>FP_NAN</i>	Not a number
<i>FP_NORMAL</i>	Normale (non zero, subnormale, infinito o NaN)
<i>FP_SUBNORMAL</i>	Subnormale
<i>FP_ZERO</i>	Zero (positivo o negativo)

isfinite La macro *isfinite* restituisce un valore diverso da zero se il suo argomento possiede un valore finito (zero, subnormale o normale, ma non infinito o NaN). La macro *isinf* restituisce un valore diverso da zero se il suo argomento possiede il valore infinito (positivo o negativo). La macro *isnan* restituisce un valore diverso da zero se il suo argomento è un valore NaN. La macro *isnormal* restituisce un valore diverso da zero se il suo argomento possiede un valore normale (che non sia: zero, subnormale, infinito o NaN).

isnan

isnormal

signbit L'ultima macro di classificazione è un po' diversa dalle altre. La macro *signbit* restituisce un valore diverso da zero se il segno del suo argomento è negativo. L'argomento non deve necessariamente essere un numero finito, questa macro funziona anche con infinito e NaN.

Funzioni trigonometriche

<i>float acosf(float x);</i>	<i>vedi acos</i>
<i>long double acosl(long double x);</i>	<i>vedi acos</i>
<i>float asinf(float x);</i>	<i>vedi asin</i>
<i>long double asinl(long double x);</i>	<i>vedi asin</i>
<i>float atanf(float x);</i>	<i>vedi atan</i>
<i>long double atanl(long double x);</i>	<i>vedi atan</i>
<i>float atan2f(float y, float x);</i>	<i>vedi atan2</i>
<i>long double atan2l(long double y,</i>	<i>vedi atan2</i>
<i> long double x);</i>	
<i>float cosf(float x);</i>	<i>vedi cos</i>
<i>long double cosl(long double x);</i>	<i>vedi cos</i>
<i>float sinf(float x);</i>	<i>vedi sin</i>
<i>long double sinl(long double x);</i>	<i>vedi sin</i>
<i>float tanf(float x);</i>	<i>vedi tan</i>
<i>long double tanl(long double x);</i>	<i>vedi tan</i>

Le uniche funzioni trigonometriche introdotte dal C99 sono analoghe alle funzioni del C89. Per una descrizione, si vedano le funzioni corrispondenti nella Sezione 23.3.

Funzioni iperboliche

<code>double acosh(double x);</code>	
<code>float acoshf(float x);</code>	
<code>long double acoshl(long double x);</code>	
<code>double asinh(double x);</code>	
<code>float asinhf(float x);</code>	
<code>long double asinhl(long double x);</code>	
<code>double atanh(double x);</code>	
<code>float atanhf(float x);</code>	
<code>long double atanhl(long double x);</code>	
<code>float coshf(float x);</code>	<i>vedi cosh</i>
<code>long double coshl(long double x);</code>	<i>vedi cosh</i>
<code>float sinh(double x);</code>	<i>vedi sinh</i>
<code>long double sinhl(long double x);</code>	<i>vedi sinh</i>
<code>float tanhf(float x);</code>	<i>vedi tanh</i>
<code>long double tanhl(long double x);</code>	<i>vedi tanh</i>

acos
asinh
atanh

Sei funzioni di questo gruppo corrispondono alle funzioni C89 cosh, sinh e tanh. Le nuove funzioni sono acosh, che calcola l'arcocoseno iperbolico, asinh che calcola l'arcoseno iperbolico e atanh che calcola l'arcotangente iperbolica.

Funzioni esponenziali e logaritmiche

<code>float expf(float x);</code>	<i>vedi exp</i>
<code>long double expl(long double x);</code>	<i>vedi exp</i>
<code>double exp2(double x);</code>	
<code>float exp2f(float x);</code>	
<code>long double exp2l(long double x);</code>	
<code>double expm1(double x);</code>	
<code>float expmf(float x);</code>	
<code>long double expml(long double x);</code>	
<code>float frexpf(float value, int *exp);</code>	<i>vedi frexp</i>
<code>long double frexpl(long double value,</code>	<i>vedi frexp</i>
<code> int *exp);</code>	
<code>int ilogb(double x);</code>	
<code>int ilogbf(float x);</code>	
<code>int ilogbl(long double x);</code>	

<code>float ldexpf(float x, int exp);</code>	<i>vedi ldexp</i>
<code>long double ldexpl(long double x, int exp);</code>	<i>vedi ldexp</i>
<code>float logf(float x);</code>	<i>vedi log</i>
<code>long double logl(long double x);</code>	<i>vedi log</i>
<code>float log10f(float x);</code>	<i>vedi log10</i>
<code>long double log10l(long double x);</code>	<i>vedi log10</i>
<code>double log1pf(double x);</code>	
<code>float log1pf(float x);</code>	
<code>long double log1pl(long double x);</code>	
<code>double log2(double x);</code>	
<code>float log2f(float x);</code>	
<code>long double log2l(long double x);</code>	
<code>double logb(double x);</code>	
<code>float logbf(float x);</code>	
<code>long double logbl(long double x);</code>	
<code>float modff(float value, float *ipti);</code>	<i>vedi modf</i>
<code>long double modfl(long double value,</code>	<i>vedi modf</i>
<code> long double *ipti);</code>	
<code>double scalbn(double x, int n);</code>	
<code>float scalbnf(float x, int n);</code>	
<code>long double scalbnl(long double x, int n);</code>	
<code>double scalbln(double x, long int n);</code>	
<code>float scalblnf(float x, long int n);</code>	
<code>long double scalblnl(long double x, long int n);</code>	
<code>long double tanl(long double x);</code>	

**exp2
expm1**

D&R

**logb
ilogb
log1p
log2**

**scalbn
scalbln**

Oltre alle nuove versioni di `exp`, `frexp`, `ldexp`, `log`, `log10` e `modf`, in questa categoria sono presenti diverse funzioni completamente nuove. Due di queste, `exp2` e `expm1`, sono delle varianti della funzione `exp`. Quando viene applicata all'argomento `x`, la funzione `exp2` restituisce $2x$, mentre la `expm1` restituisce $e^x - 1$.

La funzione `logb` restituisce l'esponente del suo argomento. Più precisamente la chiamata `logb(x)` restituisce $\log_r(|x|)$, dove r è la radice dell'aritmetica in virgola mobile (definita dalla macro `FLT_RADIX`, che tipicamente ha il valore 2). La funzione `ilogb` restituisce il valore di `logb` dopo aver applicato un cast verso il tipo `int`. La funzione `log1p` restituisce $\ln(1 + x)$ dove `x` è l'argomento della funzione stessa. La funzione `log2` calcola il logaritmo in base 2 del suo argomento.

La funzione `scalbn` restituisce il valore $x \times \text{FLT_RADIX}^n$ che viene calcolato in modo molto efficiente (non elevando esplicitamente `FLT_RADIX` all'`n`-esima potenza). La funzione `scalbln` differisce dalla `scalbn` solamente a causa del suo secondo parametro che è di tipo `long int` invece che `int`.

Funzioni di elevamento a potenza e valore assoluto

<code>double cbrt(double x);</code>	
<code>float cbrtf(float x);</code>	
<code>long double cbtrl(long double x);</code>	
<code>float fabsf(float x);</code>	<i>vedi fabs</i>
<code>long double fabsl(long double x);</code>	<i>vedi fabs</i>
<code>double hypot(double x, double y);</code>	
<code>float hypotf(float x, float y);</code>	<i>vedi sqrt</i>
<code>long double hypotl(long double x, long double y);</code>	
<code>float powf(float x, float y);</code>	<i>vedi pow</i>
<code>long double powl(long double x,</code>	
<code> long double y);</code>	<i>vedi pow</i>
<code>float sqrtf(float x);</code>	<i>vedi sqrt</i>
<code>long double sqrtl(long double x);</code>	<i>vedi sqrt</i>

Diverse funzioni di questo gruppo sono versioni aggiornate di quelle vecchie (fab, pow e sqrt). Solamente le funzioni cbrt e hypot (e le loro varianti) sono interamente nuove.

cbrt La funzione cbrt calcola la radice cubica del suo argomento. Anche la funzione pow può essere usata a tale scopo, tuttavia non è in grado di gestire argomenti negativi (verifica un errore di dominio). La funzione cbrt invece, è definita sia per argomenti positivi che per quelli negativi. Quando il suo argomento è negativo, la cbrt restituisce un risultato negativo.

hypot Quando viene applicata agli argomenti x e y , la funzione hypot restituisce $\sqrt{x^2 + y^2}$. In altre parole, questa funzione calcola l'ipotenusa di un triangolo rettangolo con lati x e y .

Funzioni di errore e gamma

<code>double erf(double x);</code>	
<code>float erff(float x);</code>	
<code>long double erfl(long double x);</code>	
<code>double erfc(double x);</code>	
<code>float erfcf(float x);</code>	
<code>long double erfcf(long double x);</code>	
<code>double lgamma(double x);</code>	
<code>float lgammaf(float x);</code>	
<code>long double lgammal(long double x);</code>	
<code>double tgamma(double x);</code>	
<code>float tgammaf(float x);</code>	
<code>long double tgammal(long double x);</code>	

erf La funzione erf calcola la **funzione di errore erf** (conosciuta anche come **funzione di errore gaussiano**), che viene utilizzata nella teoria delle probabilità, in statistica e nelle equazioni differenziali parziali. La definizione matematica di erf è:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

erfc La funzione erfc calcola la **funzione di errore complementare**, $\text{erfc}(x) = 1 - \text{erf}(x)$.

Igamma La **funzione gamma** Γ è un'estensione della funzione fattoriale che può essere applicata ai numeri reali oltre che agli interi. Quando viene applicata a un intero n la funzione restituisce $\Gamma(n) = (n - 1)!$. La definizione di Γ per i numeri non interi è più complicata. La funzione tgamma calcola Γ . La funzione lgamma calcola $\ln(|\Gamma(x)|)n$, il logaritmo naturale del valore assoluto della funzione gamma. La funzione lgamma a volte può essere più utile della stessa funzione Γ e perché questa cresce così velocemente che utilizzarla nei calcoli può produrre un overflow.



Funzioni per l'intero più vicino

<code>float ceilf(float x);</code>	<i>vedi ceil</i>
<code>long double ceill(long double x);</code>	<i>vedi ceil</i>
<code>float floorf(float x);</code>	<i>vedi floor</i>
<code>long double floorl(long double x);</code>	<i>vedi floor</i>
<code>double nearbyint(double x);</code>	
<code>float nearbyintf(float x);</code>	
<code>long double nearbyintl(long double x);</code>	
<code>double rint(double x);</code>	
<code>float rintf(float x);</code>	
<code>long double rintl(long double x);</code>	
<code>long int lrint(double x);</code>	
<code>long int lrintf(float x);</code>	
<code>long int lrintl(long double x);</code>	
<code>long long int llrint(double x);</code>	
<code>long long int llrintf(float x);</code>	
<code>long long int llrintl(long double x);</code>	
<code>double round(double x);</code>	
<code>float roundf(float x);</code>	
<code>long double roundl(long double x);</code>	

```

long int lround(double x);
long int lroundf(float x),
long int lroundl(long double x);
long long int llround(double x);
long long int llroundf(float x),
long long int llroundl(long double x);

double trunc(double x);
float truncf(float x),
long double truncl(long double x);

```

Oltre alle versioni aggiuntive delle funzioni `ceil` e `floor`, il C99 possiede un buon numero di nuove funzioni che convertono un valore a virgola mobile nell'intero più vicino. Fate attenzione quando utilizzate queste funzioni: sebbene tutte restituiscano un valore intero, alcune lo restituiscono in formato a virgola mobile (come un valore `float`, `double` o `long double`) mentre altre lo restituiscono in formato intero (come un valore `long int` o `long long int`).

- nearbyint**
`rint`
La funzione `nearbyint` arrotonda il suo argomento a un intero, restituendolo come un numero a virgola mobile. Questa funzione utilizza la corrente direzione di arrotondamento e non solleva l'eccezione floating point `inexact`. La funzione `rint` è uguale alla `nearbyint`, ma a differenza di questa solleva l'eccezione `inexact` se il risultato ha un valore diverso dall'argomento.
- lrint**
`llrint`
La funzione `lrint` arrotonda il suo argomento all'intero più vicino, in accordo alla corrente direzione di arrotondamento. La funzione `lrint` restituisce un valore `long int`. La funzione `llrint` è uguale alla `lrint` ma restituisce un valore `long long int`.
- round**
La funzione `round` arrotonda il suo argomento al valore intero più vicino, restituendolo come un numero a virgola mobile. Questa funzione arrotonda sempre verso lo zero (per esempio 3.5 viene arrotondato a 4.0).
- lround**
`llround`
La funzione `lround` arrotonda il suo argomento al valore intero più vicino, restituendolo come un valore `long int`. Come `round`, anche questa funzione arrotonda verso lo zero. La funzione `llround` è uguale alla `round`, ma a differenza di questa restituisce un valore `long long int`.
- trunc**
La funzione `trunc` arrotonda il suo argomento all'intero più vicino che non sia più grande in valore assoluto (in altre parole, tronca l'argomento verso lo zero). La funzione `trunc` restituisce il risultato come un numero a virgola mobile.

Funzioni per il resto

<code>float fmodf(float x, float y);</code>	<code>long double fmodl(long double x,</code>	<code>long double y);</code>	<i>vedi fmod</i>
<code>double remainder(double x, double y);</code>	<code>float remainderf(float x, float y);</code>	<code>long double remainderl(long double x,</code>	<i>vedi fmod</i>
<code>float remainderf(float x, float y);</code>	<code>long double remainderl(long double x,</code>	<code>long double y);</code>	

```
double remquo(double x, double y, int *quo);
float remquo(float x, float y, int *quo);
long double remquol(long double x, long double y,
                     int *quo);
```

Oltre alle versioni aggiuntive della `fmod`, questa categoria include nuove funzioni per il resto chiamate `remainder` e `remquo`.

`remainder` La funzione `remainder` restituisce $x \text{ REM } y$, dove `REM` è una funzione definita nello standard IEEE. Per $y \neq 0$, il valore di $x \text{ REM } y$ è $r = x - ny$, dove n è l'intero più vicino all'esatto valore di x/y (se x/y è a metà strada tra due interi, n è pari). Se $r = 0$, ha lo stesso valore di x .

`remquo` Quando i primi due argomenti sono uguali, la funzione `remquo` restituisce lo stesso valore della `remainder`. In aggiunta, la funzione `remquo` modifica l'oggetto puntato dal parametro `quo` in modo che questo contenga gli n bit meno significativi del quoziente intero $|x/y|$, dove n dipende dall'implementazione ma deve essere non inferiore a tre. Il valore salvato in questo oggetto sarà negativo nel caso in cui $x/y < 0$.

Funzioni di manipolazione

```
double copysign(double x, double y);
float copysignf(float x, float y),
long double copysignal(long double x, long double y);

double nan(const char *tagp),
float nanf(const char *tagp),
long double nanl(const char *tagp);

double nextafter(double x, double y),
float nextafterf(float x, float y),
long double nextafterl(long double x, long double y);

double nexttoward(double x, long double y),
float nexttowardf(float x, long double y),
long double nexttowardl(long double x,
                        long double y);
```

Le cosiddette "funzioni di manipolazione" sono tutte nuove del C99. Queste forniscono accesso ai dettagli di basso livello dei numeri a virgola mobile.

`copysign` La funzione `copysign` copia il segno di un numero in un altro. La chiamata `copysign(x, y)` restituisce un numero con il valore assoluto di x e il segno di y .

`nan` La funzione `nan` converte una stringa in un valore NaN. La chiamata `nan("n-char-sequence")` è equivalente alla `strtod("NAN(n-char-sequence)", (char**)NULL)` (guardate la discussione della funzione `strtod` [funzione `strtod` > 26.2] per una descrizione del formato `n-char-sequence`). La chiamata `nan("")` è equivalente alla chiamata `strtod("NAN()", (char**)NULL)`. Se l'argomento in una chiamata alla `nan` non possiede il valore "`n-char-sequence`" o "", la chiamata è equivalente a `strtod("NAN", (char**) NULL)`. Se i NaN tranquilli non sono supportati, la `nan` restituisce uno zero. Le chiamate alle funzioni

	nanf e nanl sono rispettivamente equivalenti alle chiamate alla strtof e alla strtold. Questa funzione viene usata per costruire un valore NaN contenente uno specifico pattern binario (ricordate che la mantissa di un valore NaN è arbitraria).
nextafter	La funzione nextafter determina il valore rappresentabile più prossimo al numero x (se tutti i valori del tipo di x fossero elencati in ordine sarebbe il numero immediatamente precedente o successivo a x). Il valore di y determina la direzione: se y < x la funzione restituisce il valore immediatamente precedente a x. Se x < y la funzione restituisce il valore immediatamente successivo. Se x e y sono uguali, la funzione nextafter restituisce y.
D&R	
nexttoward	La funzione nexttoward è uguale alla nextafter ma differisce da questa per il fatto che il parametro y è di tipo long double e non double. Se x e y sono uguali la funzione restituisce y convertito nel tipo restituito dalla funzione stessa. Il vantaggio della nexttoward è che il valore di un qualsiasi tipo a virgola mobile (reale) può essere passato come secondo argomento senza il pericolo che questo venga erroneamente convertito in un tipo più piccolo.

Funzioni di massimo, minimo e differenza positiva

```
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);

double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

fdim La funzione fdim calcola la differenza positiva tra x e y:

$$\begin{cases} x - y & \text{if } x > y \\ +0 & \text{if } x \leq y \end{cases}$$

fmax
fmin La funzione fmax restituisce il più grande tra i suoi argomenti, mentre la fmin restituisce il minore.

Moltiplicazione e somma in virgola mobile

```
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y,
                 long double z);
```

fma La funzione fma moltiplica i suoi primi argomenti e poi somma il terzo argomento. In altre parole, possiamo sostituire l'istruzione

a = b * c + d;

con

```
a = fma(b, c, d);
```

Questa funzione è stata aggiunta al C99 perché alcune nuove CPU possiedono un'istruzione detta *fused multiply-add* che esegue sia la moltiplicazione che la somma. Chiamare la funzione *fma* dice al compilatore di utilizzare questa istruzione (se disponibile), la quale può essere più veloce rispetto all'esecuzione di istruzioni separate di moltiplicazione e somma. Inoltre l'istruzione *fused multiply-add* effettua una sola operazione di arrotondamento e non due, e quindi può produrre un risultato più accurato. Questa istruzione è particolarmente utile per gli algoritmi che eseguono una serie di moltiplicazioni e somme, come quelli per il prodotto scalare di due vettori o la moltiplicazione tra due matrici.

Per determinare se una chiamata alla funzione *fma* è effettivamente una buona idea, un programma C99 può controllare se è definita la macro *FP_FAST_FMA*. Se questa macro è definita, chiamare la funzione *fma* dovrebbe essere più veloce (o almeno della stessa velocità) che effettuare delle operazioni di moltiplicazione e somma separate. Le macro *FP_FAST_FMAF* e *FP_FAST_FMAL* giocano lo stesso ruolo rispettivamente delle funzioni *fmaf* e *fmal*.

Eseguire una moltiplicazione e una somma combinate è un esempio di quello che lo standard C99 chiama "contrazione", dove due o più operazioni matematiche vengono combinate assieme ed eseguite come una singola operazione. Come abbiamo visto per la funzione *fma*, spesso la contrazione porta a una migliore velocità e una maggiore accuratezza. Tuttavia i programmatore potrebbero voler controllare se la contrazione viene fatta automaticamente (in opposizione alle chiamate alla funzione *fma* che sono delle richieste esplicite per la contrazione), dato che può condurre a risultati leggermente diversi. In casi estremi la contrazione può evitare un'eccezione floating point che altrimenti sarebbe stata sollevata.

 Il C99 fornisce una direttiva pragma [[direttive pragma > 14.5](#)] chiamata *FP_CONTRACT* che fornisce al programmatore il controllo sulla contrazione. Ecco come viene usata questa direttiva:

```
#pragma STDC FP_CONTRACT on-off-switch
```

Il valore di *on-off-switch* può essere sia ON, OFF o DEFAULT. Se viene selezionato ON, al compilatore è permesso contrarre le espressioni. Se è selezionato OFF, al compilatore viene proibito di contrarre le espressioni. Il valore DEFAULT è utile per ripristinare le impostazioni di default (che possono essere sia ON che OFF). Se la direttiva pragma viene usata al livello più esterno di un programma (al di fuori di tutte le funzioni), rimane effettiva fino a quando non compare una successiva direttiva pragma *FP_CONTRACT* oppure fino alla fine del file. Se la direttiva viene usata all'interno di un'istruzione composta (incluso il corpo di una funzione), questa deve comparire all'inizio, prima di ogni dichiarazione o istruzione. In tal caso la direttiva rimarrà valida fino alla fine dell'istruzione composta, a meno che non venga annullata da un'altra direttiva pragma. Un programma può ancora chiamare la funzione *fma* per eseguire una contrazione esplicita anche quando la direttiva *FP_CONTRACT* è stata utilizzata per proibire la contrazione automatica delle espressioni.

Macro per i confronti

```
int isgreater(real-floating x, real-floating y),
int isgreaterequal(real-floating x, real-floating y),
int isless(real-floating x, real-floating y),
int islessequal(real-floating x, real-floating y),
int islessgreater(real-floating x, real-floating y),
int isunordered(real-floating x, real-floating y);
```

La nostra ultima categoria consiste di macro simili a funzioni che confrontano due numeri. Queste macro sono state progettate per accettare argomenti di qualsiasi tipo reale a virgola mobile.

Le macro per i confronti esistono a causa di un problema che può sorgere quando dei numeri a virgola mobile vengono confrontati utilizzando i normali operatori relazionali come < e >. Se uno degli operandi (o entrambi) è un NaN, un confronto di questo tipo può causare il sollevarsi dell'eccezione floating point *invalid*. Questo perché i valori NaN, a differenza degli altri valori a virgola mobile, sono considerati non ordinabili. Le macro di confronto possono essere utilizzate per evitare questa eccezione. Queste macro vengono dette versioni "tranquille" degli operatori relazionali perché effettuano il loro compito senza sollevare un'eccezione.

Le macro `isgreater`, `isgreaterequal`, `isless` e `islessequal` eseguono le stesse operazioni rispettivamente degli operatori `>`, `>=`, `<` e `>=`. A differenza degli operatori però non sollevano l'eccezione floating point *invalid* nel caso in cui gli argomenti non fossero ordinabili.

La chiamata `islessgreater(x, y)` è equivalente a `(x) < (y) || (x) > (y)` a eccezione del fatto che non calcola due volte il valore di `x` e `y`, e, come le macro precedenti, non solleva un'eccezione *invalid* nel caso `x` o `y` non fossero ordinabili.

La macro `isunordered` restituisce il valore 1 se i suoi argomenti non sono ordinabili (almeno uno di essi è NaN), negli altri casi restituisce il valore 0.

23.5 L'header `<ctype.h>` (C99): gestione dei caratteri

L'header `<ctype.h>` fornisce due tipi di funzioni: le funzioni di classificazione dei caratteri (come la `isdigit`, che controlla se un carattere corrisponde a una cifra) e le funzioni di *case-mapping* (come la `toupper`, la quale converte una lettera minuscola in una maiuscola).

Sebbene il C non ci imponga di usare le funzioni appartenenti a `<ctype.h>` per controllare ed eseguire conversioni di case, sfruttarle è una buona idea. Per prima cosa queste funzioni sono state ottimizzate in fatto di velocità (infatti molte sono implementate come macro). Secondariamente otterremo un programma più portabile visto che queste funzioni lavorano con qualsiasi set di caratteri. Inoltre le funzioni di `<ctype.h>` adattano il loro comportamento quando la localizzazione viene modificata [**localizzazione > 25.1**], il che ci aiuta a scrivere programmi che funzionano a dovere in diverse parti del mondo.

Le funzioni presenti in <ctype.h> accettano tutti argomenti di tipo int e restituiscono valore di tipo int. In molti casi l'argomento è già memorizzato in una variabile di tipo int (perché spesso viene letto con una chiamata alle funzioni fgetc, getc o getchar). Se invece l'argomento è di tipo char, dovremo fare attenzione. Il C può convertire automaticamente un argomento char al tipo int. Se char è un tipo senza segno oppure stiamo usando un set di caratteri a sette bit come l'ASCII, la conversione procederà senza problemi. Se invece è un tipo con segno e se alcuni caratteri richiedono otto bit, allora la conversione di questo tipo di caratteri da char a int avrà un esito negativo. Il comportamento delle funzioni presenti in <ctype.h> non è definito per argomenti negativi (diversi da EOF) e può essere una potenziale causa di problemi. In una situazione del genere, per sicurezza è meglio effettuare un cast dell'argomento al tipo unsigned char (per una portabilità massima alcuni programmatore effettuano sempre il cast di un valore char al tipo unsigned char quando lo passano a una funzione <ctype.h>).

Funzioni per la classificazione dei caratteri

```
int isalnum(int c);
int isalpha(int c);
int isblank(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
```

Ogni funzione per la classificazione dei caratteri restituisce un valore diverso da zero e il suo argomento possiede una particolare proprietà. La Tabella 23.10 elenca la proprietà che viene controllata da ognuna di queste funzioni.

Tabella 23.10 Funzioni per la classificazione dei caratteri

Nome	Controllo
isalnum(c)	c è un carattere alfanumerico?
isalpha(c)	c è una lettera?
isblank(c)	c è un carattere vuoto? ^t
iscntrl(c)	c è un carattere di controllo? ^{tt}
isdigit(c)	c è una cifra decimale?

continua

Nome	Controllo
isgraph(c)	c è un carattere stampabile (diverso da uno spazio)?
islower(c)	c è una lettera minuscola?
isprint(c)	c è un carattere stampabile (incluso lo spazio)?
ispunct(c)	c è un carattere di punteggiatura? ^{***}
isspace(c)	c è un carattere di spazio bianco? ^{****}
isupper(c)	c è una lettera maiuscola?
isxdigit(c)	c è una cifra esadecimale?

^{*}I caratteri vuoti standard sono lo spazio e la tabulazione orizzontale (\t). Questa funzione è nuova del C99.

^{**}Nello standard ASCII i caratteri di controllo sono quelli che vanno dal codice \x00 al \x1f oltre che il carattere \x7f.

^{***}Tutti i caratteri stampabili ad eccezione di quelli per i quali isspace e isalnum sono vere, sono considerati punteggiatura.

^{****}I caratteri di spazio bianco sono: lo spazio, form-feed (\f), new-line (\n), carriage-return (\r), tab orizzontale (\t) e tab verticale (\v).



La definizione del C99 per la funzione ispunct è leggermente diversa da quella del C89. Nel C89 ispunct(c) controlla se c è un carattere stampabile che non sia un spazio o un carattere per il quale il valore di isalnum(c) è true. Nel C99, ispunct(c) controlla se c è un carattere stampabile per il quale ne il valore di isspace(c) ne il valore di isalnum(c) sono pari a true.

PROGRAMMA

Testare le funzioni di classificazione dei caratteri

Il programma seguente è un dimostratore per le funzioni di classificazione dei caratteri (ad eccezione della funzione isblank che è nuova del C99). Il programma applicherà queste funzioni ai caratteri presenti nella stringa "azAZO !\t".

```
/* Testa le funzioni per la classificazione dei caratteri */

#include <ctype.h>
#include <stdio.h>

#define TEST(f) printf(" %c ", f(*p) ? 'x' : ' ')

int main(void)
{
    char *p;

    printf("  alnum      cntrl      graph      print"
           "  space      xdigit\n"
           "  alpha       digit      lower      punct"
           "  upper\n");

    for (p = "azAZO !\t"; *p != '\0'; p++) {
        if (iscntrl(*p))
            printf("\\"x%02x:", *p);
        else
            printf("  %c:", *p);
    }
}
```

```

    TEST(isalnum);
    TEST(isalpha);
    TEST(iscntrl);
    TEST(isdigit);
    TEST(isgraph);
    TEST(islower);
    TEST(isprint);
    TEST(ispunct);
    TEST(isspace);
    TEST(isupper);
    TEST(isxdigit);
    printf("\n");
}
return 0;
}

```

Il programma produce il seguente output:

	alnum	cntrl	graph	print	space	xdigit
	alpha	digit	lower	punct	upper	
a:	x	x		x	x	x
z:	x	x		x	x	
A:	x	x		x		x
Z:	x	x		x		x
O:	x		x	x		x
:				x		x
!:			x	x	x	
\x09:		x			x	

Funzioni per il case-mapping

```

int tolower(int c);
int toupper(int c);

```

tolower toupper La funzione toupper restituisce una versione minuscola della lettera che le viene passata come argomento, mentre la toupper restituisce una versione maiuscola. Se l'argomento di queste funzioni non corrisponde a una lettera, queste lo restituiscono senza modificarlo.

PROGRAMMA Testare le funzioni di case-mapping

Il programma seguente applica le funzioni di case-mapping ai caratteri presenti nella stringa "aA0!".

```

tcasemap.c /* Test per le funzioni di case-mapping */

#include <ctype.h>
#include <stdio.h>

```

```

int main(void)
{
    char *p;

    for (p = "aA0!"; *p != '\0'; p++) {
        printf("tolower('%c') is '%c'; ", *p, tolower(*p));
        printf("toupper('%c') is '%c'\n", *p, toupper(*p));
    }
    return 0;
}

```

Il programma produce il seguente output:

```

tolower('a') is 'a'; toupper('a') is 'A'
tolower('A') is 'a'; toupper('A') is 'A'
tolower('0') is '0'; toupper('0') is '0'
tolower('!') is '!'; toupper('!') is '!'

```

23.6 L'header <string.h> (C99): manipolazione delle stringhe

Abbiamo incontrato per la prima volta l'header <string.h> nella Sezione 13.5. Quella sezione trattava le operazioni più basilari sulle stringhe: copiarle, concatenarle, confrontarle oltre che trovare la lunghezza di una stringa. Come vedremo tra poco, nell'header <string.h> ci sono diverse funzioni per la manipolazione delle stringhe, così come funzioni che operano sui vettori di caratteri che non finiscono necessariamente con un null. Le funzioni di quest'ultima categoria hanno nomi che iniziano con `mem` per suggerire che gestiscono blocchi di memoria invece che stringhe. Questi blocchi di memoria possono contenere dei dati di qualsiasi tipo, per questo gli argomenti delle funzioni `mem` sono di tipo `void *` invece che `char *`.

L'header <string.h> fornisce cinque tipi di funzioni.

- **Funzioni per la copia.** Funzioni che copiano i caratteri da un punto della memoria ad un altro.
- **Funzioni di concatenamento.** Funzioni che sommano caratteri alla fine della stringa.
- **Funzioni di confronto.** Funzioni che confrontano vettori di caratteri.
- **Funzioni di ricerca.** Funzioni che vanno alla ricerca di un particolare carattere all'interno di un vettore, un insieme di caratteri o una stringa.
- **Funzioni varie.** Funzioni che inizializzano un blocco di memoria o calcolano la lunghezza di una stringa.

Discuteremo ora di queste funzioni, esaminandole un gruppo alla volta.

Funzioni per la copia

```
void *memcpy(void * restrict s1,
            const void * restrict s2, size_t n);
void *memmove(void * restrict s1, const void * restrict s2, size_t n);
char *strcpy(char * restrict s1,
            const char * restrict s2);
char *strncpy(char * restrict s1,
              const char * restrict s2, size_t n);
```



`memcpy`
`memmove`

`strcpy`
`strncpy`

Le funzioni di questa categoria copiano i caratteri (byte) da una locazione della memoria (la "sorgente") a un'altra (la "destinazione"). Ogni funzione richiede che il primo argomento punti alla destinazione e che il secondo punti alla sorgente. Tutte le funzioni di copia restituiscono il primo argomento (un puntatore alla destinazione). La funzione `memcpy` copia `n` caratteri dalla sorgente alla destinazione, dove `n` è il terzo argomento della funzione. Se la sorgente e la destinazione si sovrappongono il comportamento è indefinito. La funzione `memmove` è uguale alla `memcpy`, ma a differenza di questa funziona correttamente anche quando la sorgente e la destinazione si sovrappongono.

La funzione `strcpy` copia una stringa terminante con null dalla sorgente alla destinazione. La funzione `strncpy` è simile alla `strcpy`, ma a differenza di quest'ultima non copia più di `n` caratteri, dove `n` è il terzo argomento della funzione (nel caso `n` fosse troppo piccolo, la funzione non sarà in grado di copiare il carattere null che segna il termine della stringa). Se incontra un carattere null nella sorgente, la funzione `strncpy` aggiunge alla destinazione dei caratteri null fino a quando non ha scritto un totale di `n` caratteri. Per le funzioni `strcpy` e `strncpy`, come per la `memcpy`, il funzionamento non è garantito nel caso sorgente e destinazione si sovrapponessero.

Gli esempi seguenti illustrano le funzioni di copia e i commenti mostrano come vengono effettivamente copiati i caratteri.

```
char source[] = {'h', 'o', 't', '\0', 't', 'e', 'a'};
char dest[];

memcpy(dest, source, 3);      /* h, o, t          */
memcpy(dest, source, 4);      /* h, o, t, \0      */
memcpy(dest, source, 7);      /* h, o, t, \0, t, e, a */

memmove(dest, source, 3);     /* h, o, t          */
memmove(dest, source, 4);     /* h, o, t, \0      */
memmove(dest, source, 7);     /* h, o, t, \0, t, e, a */

strcpy(dest, source);        /* h, o, t, \0      */
strncpy(dest, source, 3);     /* h, o, t          */
strncpy(dest, source, 4);     /* h, o, t, \0      */
strncpy(dest, source, 7);     /* h, o, t, \0, \0, \0 */
```

Osservate che le funzioni `memcpy`, `memmove` e `strncpy` non richiedono una stringa terminante con il carattere null. Queste funzioni sono in grado di gestire un qualsiasi

blocco di memoria. La funzione strcpy, d'altro canto, non smette di copiare fino a quando non raggiunge il carattere null e quindi funziona solamente con le stringhe terminanti con null.

La Sezione 13.5 forniva degli esempi di come la strcpy e la strncpy vengano tipicamente utilizzate. Sebbene nessuna delle due funzioni sia completamente sicura, la strncpy quanto meno dà un modo per limitare il numero di caratteri che verranno copiati.

Funzioni per il concatenamento

```
char *strcat(char * restrict s1,
             const char * restrict s2);
char *strncat(char * restrict s1,
              const char * restrict s2, size_t n);
```

strcat La funzione strcat accoda il suo secondo argomento alla fine del primo argomento. Entrambi gli argomenti devono essere delle stringhe terminanti con il carattere null. La funzione strcat mette un carattere null alla fine della stringa concatenata. Considerate l'esempio seguente:

```
char str[7] = "tea";
strcat(str, "bag"); /* aggiunge b, a, g, \0 alla fine di str */
```

La lettera b sovrascrive il carattere null posto dopo la a presente in "tea". Ora la variabile str contiene la stringa "teabag". La funzione strcat restituisce il suo primo argomento (un puntatore).

strncat La funzione strncat è uguale alla strcat a eccezione del fatto che il suo terzo argomento pone un limite al numero di caratteri che verranno copiati:

```
char str[7] = "tea";
strncat(str, "bag", 2); /* aggiunge b, a, \0 a str */
strncat(str, "bag", 3); /* aggiunge b, a, g, \0 a str */
strncat(str, "bag", 4); /* aggiunge b, a, g, \0 a str */
```

Tutti questi esempi mostrano che, con la funzione strncat, la stringa risultante termina sempre con il carattere null.

Nella Sezione 13.5 abbiamo visto che una chiamata alla strncat presenta sempre il seguente aspetto:

```
strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
```

Il terzo argomento calcola la quantità di spazio rimanente in str1 (dato dall'espressione sizeof(str1) - strlen(str1)) e poi sottrae 1 per assicurare che ci sia spazio per il carattere null.

Funzioni di confronto

```
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2,
            size_t n);
size_t strxfrm(char * restrict s1,
               const char * restrict s2, size_t n);
```

Le funzioni di confronto si dividono in due gruppi. Le funzioni del primo gruppo (`memcmp`, `strcmp` e `strncmp`) confrontano il contenuto di due vettori di caratteri. Le funzioni del secondo gruppo (`strcoll` e `strxfrm`) vengono usate quando si deve tenere conto della localizzazione [localizzazione > 25.1].

memcmp
strcmp
strncmp

Le funzioni `memcmp`, `strcmp` e `strncmp` hanno molto in comune. Tutte e tre si aspettano che vengano passati dei puntatori a vettori di caratteri. I caratteri del primo vettore vengono confrontati uno a uno con quelli del secondo vettore. Tutte e tre le funzioni terminano non appena viene rilevata un'incongruenza. Inoltre le tre funzioni restituiscono un intero negativo, il valore zero o un intero positivo a seconda che il carattere del primo vettore che ha fermato il processo di scansione sia rispettivamente minore, uguale o maggiore di quello corrispondente nel secondo vettore.

Le diversità tra le tre funzioni hanno a che fare con il termine delle operazioni di confronto dei caratteri nel caso in cui non venissero rilevate differenze. Alla funzione `memcmp` viene passato un terzo argomento, `n`, che limita il numero di confronti che devono essere eseguiti. La `memcmp`, inoltre, non presta particolare attenzione ai caratteri null. La `strcmp` non ha un limite predeterminato ma termina il confronto quando raggiunge un carattere null in uno dei due vettori (ne risulta che la `strcmp` funziona solamente con le stringhe che terminano con il carattere null). La funzione `strncmp` fonde assieme la `memcmp` e la `strcmp`. Questa funzione si ferma dopo aver eseguito `n` confronti o se viene raggiunto un carattere null in uno dei due vettori.

Gli esempi seguenti illustrano le funzioni `memcmp`, `strcmp` e `strncmp`:

```
char s1[] = {'b', 'i', 'g', '\0', 'c', 'a', 'r'};
char s2[] = {'b', 'i', 'g', '\0', 'c', 'a', 't'};

if (memcmp(s1, s2, 3) == 0) /* true */
if (memcmp(s1, s2, 4) == 0) /* true */
if (memcmp(s1, s2, 7) == 0) /* false */

if (strcmp(s1, s2) == 0) /* true */
if (strncmp(s1, s2, 3) == 0) /* true */
if (strncmp(s1, s2, 4) == 0) /* true */
if (strncmp(s1, s2, 7) == 0) /* true */
```

strcoll

La funzione `strcoll` è simile alla `strcmp` ma il risultato del confronto dipende dalla localizzazione corrente.

strxfrm

La maggior parte delle volte, la funzione `strcoll` è adeguata per eseguire dei confronti dipendenti dalla localizzazione. Occasionalmente, però, potremmo aver bisogno di

eseguire il confronto più di una volta (potrebbe essere un problema visto che `strcoll` non è particolarmente veloce) oppure modificare la localizzazione senza effetti sul risultato del confronto. In queste situazioni si può utilizzare la funzione `strxfrm` (*string trasform*) in alternativa alla `strcoll`.

La `strxfrm` trasforma il suo secondo argomento (una stringa) ponendo il carattere puntato dal primo argomento. Il terzo argomento limita il numero di caratteri scritti nel vettore, incluso il carattere null di termine. Chiamare la `strcmpl` su due stringhe trasformate deve produrre lo stesso risultato (negativo, zero o positivo) ottenuto chiamando la `strcoll` con le stringhe originali.

La funzione `strxfrm` restituisce la lunghezza della stringa trasformata. Ne risulta che tipicamente questa funzione viene chiamata due volte: una volta per determinare la lunghezza della stringa da trasformare e una per eseguire la trasformazione. Ecco un esempio:

```
size_t len;
char *transformed;

len = strxfrm(NULL, original, 0);
transformed = malloc(len + 1);
strxfrm(transformed, original, len);
```

Funzioni di ricerca

```
void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char *strpbrk(const char *s1, const char *s2);
char *strchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char * restrict s1,
            const char * restrict s2);
```

strchr La funzione `strchr` ricerca un particolare carattere all'interno di una stringa. Il codice seguente illustra come possiamo utilizzare la `strchr` per cercare la lettera `f` all'interno di una stringa.

```
char *p, str[] = "Form follows function.";
p = strchr(str, 'f'); /* trova la prima 'f' */
```

La `strchr` restituisce un puntatore alla prima occorrenza di `f` all'interno di `str` (nella parola `follows`). Localizzare occorrenze multiple di un carattere è semplicemente un esempio, la chiamata

```
p = strchr(p + 1, 'f'); /* trova la prossima 'f' */
```

trova la seconda `f` presente in `str` (quella nella parola `function`). Se non è in grado di trovare il carattere desiderato, la `strchr` restituisce un puntatore nullo.

- memchr La funzione `memchr` è simile alla `strchr` ma smette di cercare dopo un certo numero specificato di caratteri invece di fermarsi al primo carattere null. Il terzo argomento della `memchr` limita il numero di caratteri che questa può esaminare. Questa è una caratteristica utile se non vogliamo effettuare la ricerca nell'intera stringa o se stiamo effettuando le ricerche in un blocco di memoria che non termina necessariamente con il carattere null. L'esempio seguente utilizza la `memchr` per effettuare le ricerche su un vettore di caratteri che è sprovvisto del carattere null alla fine:
- ```
char *p, str[22] = "Form follows function.";
p = memchr(str, 'f', sizeof(str));
```
- Come la funzione `strchr` anche la `memchr` restituisce un puntatore alla prima occorrenza del carattere. Se non è in grado di trovare il carattere desiderato, questa funzione restituisce un puntatore nullo.
- strrchr La funzione `strrchr` è simile alla `strchr` ma effettua la ricerca all'interno della stringa in ordine *inverso*:
- ```
char *p, str[] = "Form follows function.";
p = strrchr(str, 'f'); /* trova l'ultima 'f' */
```
- In questo esempio, per prima cosa la `strrchr` cerca il carattere null posto alla fine della stringa e poi procede a ritroso per localizzare la lettera f (quella della parola `function`). Come la `strchr` e la `memchr`, anche la `strrchr` restituisce un puntatore nullo nel caso in cui non trovasse il carattere desiderato. La funzione `strpbrk` è più generale della `strchr`: infatti restituisce un puntatore al carattere che si trova più a sinistra nel primo argomento e che corrisponde a un *qualsiasi* carattere presente nel secondo argomento:
- ```
char *p, str[] = "Form follows function.";
p = strpbrk(str, "mn"); /* cerca la prima 'm' o 'n' */
```
- In questo esempio `p` punterà alla prima lettera `m` della parola `Form`. La `strpbrk` restituisce un puntatore nullo nel caso in cui non trovasse corrispondenze.
- strspn strcspn D&R Le funzioni `strspn` e `strcspn`, diversamente dalle altre funzioni di ricerca, restituiscono un intero (di tipo `size_t`) rappresentante una posizione all'interno della stringa. Quando viene passata una stringa all'interno della quale cercare un insieme di caratteri, la funzione `strspn` restituisce l'indice del primo carattere che *non appartiene* all'insieme. Quando vengono passati degli argomenti simili, la `strcspn` restituisce l'indice del primo carattere che *appartiene* all'insieme. Ecco un esempio di entrambe le funzioni:
- ```
size_t n;
char str[] = "Form follows function.";

n = strspn(str, "morF"); /* n = 4 */
n = strspn(str, "\t\n"); /* n = 0 */
n = strcspn(str, "morF"); /* n = 0 */
n = strcspn(str, "\t\n"); /* n = 4 */
```

strstr La funzione strstr cerca all'interno del suo primo argomento (una stringa) una rispondenza con il suo secondo argomento (anch'esso una stringa). Nell'esempio seguente, la strstr cerca la parola fun:

```
char *p, str[] = "Form follows function.";
p = strstr(str, "fun"); /* cerca "fun" in str */
```

La strstr restituisce un puntatore alla prima occorrenza presente nella stringa ricerca. Nel caso non trovasse la stringa la funzione restituirebbe un puntatore nullo. Dopo la chiamata presentata nell'esempio, la variabile p punterà alla lettera f della parola function.

strtok La funzione strtok è la più complicata delle funzioni di ricerca. È stata progettata per cercare all'interno di una stringa la presenza di un *token* (una sequenza di caratteri che non includono certi caratteri di delimitazione). La chiamata strtok(s1, s2) scansiona la stringa s1 cercando una sequenza non vuota di caratteri che *non* appartengano alla stringa s2. La funzione segnala la fine del token ponendo in s1 il carattere null immediatamente dopo l'ultimo carattere del token stesso. La funzione, inoltre, restituisce un puntatore al primo carattere presente nel token.

Cosa rende strtok particolarmente utile è che le chiamate successive possono trovare ulteriori token all'interno della stessa stringa. La chiamata strtok(NULL, s2) continua la ricerca iniziata dalla chiamata precedente alla funzione strtok. Come prima, la s2 segnala la fine del token con il carattere null e poi restituisce un puntatore all'inizio del token stesso. Il processo può essere ripetuto fino a quando la strtok restituisce un puntatore nullo, indicando in questo modo che il token non è stato trovato.

Per vedere come funziona la strtok, la useremo per estrarre il mese, il giorno e l'anno da una data scritta nel formato

mese giorno, anno

dove degli spazi e/o delle tabulazioni separano il mese dal giorno e il giorno dall'anno. Inoltre gli spazi e le tabulazioni possono anche precedere la virgola. Supponiamo che inizialmente la stringa str possieda il seguente aspetto:

str	[]	A	p	r	i	l	[]	[]	2	8	,	1	9	9	8	\0
-----	-----	---	---	---	---	---	-----	-----	---	---	---	---	---	---	---	----

Dopo la chiamata

```
p = strtok(str, " \t");
```

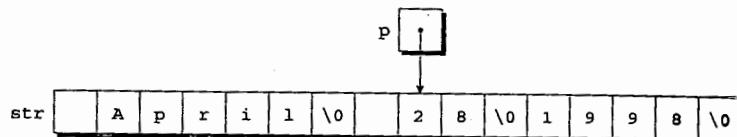
str avrà questo aspetto:

p	[]	A	p	r	i	l	\0	[]	[]	2	8	,	1	9	9	8	\0
---	-----	---	---	---	---	---	----	-----	-----	---	---	---	---	---	---	---	----

La variabile p punta al primo carattere nella stringa contenente il mese, la quale a termine con il carattere null. Chiamare la strtok con un puntatore nullo come primo argomento fa sì che questa riprenda la ricerca da dove l'aveva interrotta:

`p = strtok(NULL, " \t,");`

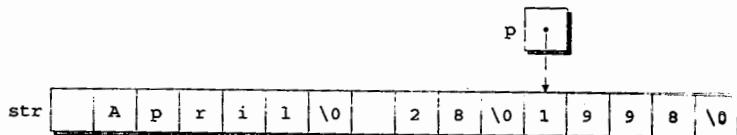
Dopo questa chiamata, `p` punta al primo carattere del giorno:



Un'ultima chiamata alla funzione `strtok` individua l'anno:

`p = strtok(NULL, " \t");`

Dopo questa chiamata `str` si presenterà in questo modo:



Quando la funzione `strtok` viene chiamata ripetutamente al fine di scomporre una stringa in token, il secondo argomento non deve essere necessariamente lo stesso ogni chiamata. Nel nostro esempio la seconda chiamata alla `strtok` aveva l'argomento `" \t,"` invece che `" \t"`.

La funzione `strtok` presenta diversi ben noti problemi che limitano il suo utilizzo. Ne menzioneremo solamente un paio. Per prima cosa la funzione opera solo su una stringa alla volta, non è in grado di condurre ricerche simultanee su due stringhe diverse. Inoltre la `strtok` tratta una sequenza di delimitatori nello stesso modo in cui tratta un singolo delimitatore, il che la rende inadatta per le applicazioni nelle quali una stringa contiene una serie di campi separati da un delimitatore (come una virgola) e alcuni di questi campi sono vuoti.

Funzioni varie

```
void *memset(void *s, int c, size_t n);
size_t strlen(const char *s);
```

`memset` La funzione `memset` salva copie multiple di un carattere in una specifica area di memoria. Se `p` punta a un blocco di `N` byte, per esempio, la chiamata

`memset(p, ' ', N);`

salverà uno spazio in ogni byte presente nel blocco. Uno degli usi di questa funzione è quello di inizializzare un vettore con i bit a zero:

`memset(a, 0, sizeof(a));`

La funzione `memset` restituisce il suo primo argomento (un puntatore).

strlen La funzione `strlen` restituisce la lunghezza di una stringa non contando il carattere `null`. Guardate la Sezione 13.5 per avere degli esempi di chiamata alla `strlen`.

Esiste un'altra funzione per le stringhe, la `strerror` [funzione `strerror` > 24.2], ma però verrà trattata assieme all'header `<errno.h>`.

Domande & Risposte

D: Perché esiste la funzione `expm1`? Alla fine tutto quello che fa è sottrarre dal valore restituito dalla funzione `exp`. [p. 625]

R: Quando applicata ai numeri che sono prossimi allo zero, la funzione `exp` restituisce un valore che è molto vicino a 1. Il risultato ottenuto sottraendo 1 dal valore restituito dalla `exp` potrebbe non essere accurato a causa dell'errore di arrotondamento. In questa situazione la funzione `expm1` fornisce un risultato più accurato.

La funzione `log1p` esiste per ragioni simili. Per i valori di `x` che sono prossimi allo zero, il valore di `log1p(x)` dovrebbe essere più accurato di quello di `log(1 + x)`.

D: Perché la funzione che calcola la funzione gamma viene chiamata `tgamma` invece di essere chiamata semplicemente `gamma`? [p. 627]

R: Nel momento in cui lo standard C99 venne scritto, alcuni compilatori fornivano una funzione chiamata `gamma`, ma questa calcolava il logaritmo delle funzioni gamma. Il nome di questa funzione venne successivamente modificato in `lgamma`. Scegliere il nome `gamma` per la funzione gamma sarebbe andato in conflitto con l'uso esistente. Di conseguenza il comitato del C99 decise di usare al suo posto il nome `tgamma` (*time gamma*).

D: Perché la descrizione della funzione `nextafter` dice che se `x` e `y` sono uguali questa restituisce `y`? Se `x` e `y` sono uguali, qual è la differenza nel risultato `x` o `y`? [p. 630]

R: Considerate la chiamata `nextafter(-0.0, +0.0)`, nella quale gli argomenti sono matematicamente uguali. Restituendo `y` invece di `x`, la funzione ha un valore restituito pari a `+0.0` (invece che `-0.0` che sarebbe contro intuitivo). Similmente la chiamata `nextafter(+0.0, -0.0)` restituisce `-0.0`.

D: Perché l'header `<string.h>` fornisce così tanti modi per fare la stessa cosa? Abbiamo veramente bisogno di quattro funzioni per la copia (`memcpy`, `memmove`, `strcpy` e `strncpy`)? [p. 637]

R: Iniziamo con `memcpy` e `strcpy`. Queste funzioni vengono utilizzate per scopi differenti. La `strcpy` può copiare solo un vettore di caratteri che termini con il carattere `null` (in altre parole una stringa), mentre la `memcpy` è in grado di copiare un blocco di memoria che non possiede questo terminatore (un vettore di interi per esempio).

Le altre funzioni ci permettono di scegliere tra sicurezza e performance. La `strncpy` è più sicura della `strcpy` visto che limita il numero di caratteri che possono essere copiati. Tuttavia paghiamo un prezzo per questa sicurezza dato che la `strncpy` probabilmente sarà più lenta della `strcpy`. Usare `memmove` comporta un simile trade-off: la funzione `memmove` copia i byte da una regione della memoria a un'altra che potrebbe sovrapporsi alla prima. Il funzionamento della `memcpy` non è garantito in simili circostanze.

costanze. Tuttavia se possiamo garantire che le due regioni non si sovrappongono, la funzione `memcpy` si rivelerà più veloce della `memmove`.

D: Perché la funzione `strspn` possiede un nome così strano? [p. 641]

R: Invece di pensare al valore restituito dalla `strspn` come all'indice del primo carattere che *non* appartiene all'insieme specificato, possiamo pensarla come alla lunghezza del più lungo *span* di caratteri che *appartengono* all'insieme.

Esercizi

Sezione 23.3



1. Estendete la funzione `round_nearest` in modo che arrotondi un numero in virgola mobile x all' n -esima cifra decimale. Per esempio, la chiamata `round_nearest(3.14159, 3)` restituirebbe il valore 3.142. *Suggerimento:* moltiplicate x per 10^n , arrotondate all'intero più vicino e poi dividete per 10^n . Assicuratevi che la vostra funzione operi correttamente sia per i valori positivi di x che per quelli negativi.

Sezione 23.4

2. Scrivete la seguente funzione:

```
double evaluate_polynomial(double a[], int n, double x);
```

La funzione dovrà restituire il valore del polinomio $a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$, dove i coefficienti a_i sono contenuti negli elementi corrispondenti del vettore a di lunghezza $n + 1$. Per calcolare il valore del polinomio utilizzate la regola di Horner:

$$((\dots((a_n x + a_{n-1}) x + a_{n-2}) x + \dots) x + a_1) x + a_0$$

Utilizzate la funzione `fma` per eseguire le moltiplicazioni e le addizioni.

C99

3. Controllate la documentazione del vostro compilatore per capire se effettua la contrazione delle operazioni aritmetiche e, in tal caso, in quali circostanze questo viene fatto.

Sezione 23.5

4. Utilizzando le funzioni `isalpha` e `isalnum`, scrivete una funzione che controlli se una stringa segue la sintassi di un identificatore C (consiste di lettere, cifre e caratteri underscore, con una lettera o un underscore all'inizio).
5. Usando la funzione `isxdigit`, scrivete una funzione che controlli se una stringa rappresenta un numero esadecimale valido (consiste solamente di cifre esadecimali). In tal caso, la funzione dovrà restituire il valore del numero sotto forma di `long int`. Altrimenti la funzione dovrà restituire -1.

Sezione 23.6



6. In ognuno di questi casi, dite quale funzione sarebbe più indicata: `memcpy`, `memmove`, `strcpy` o `strncpy`. Assumete che l'azione indicata debba essere eseguita con una singola chiamata.
 - (a) Traslate gli elementi di un vettore di una posizione "verso il basso" in modo da lasciare spazio per un nuovo elemento nella posizione 0.
 - (b) Cancellare il primo carattere di una stringa terminante con il carattere null spostando indietro tutti i caratteri di una posizione.

- (c) Copiare una stringa in un vettore di caratteri che potrebbe non essere sufficientemente grande per contenerla. Se il vettore è troppo piccolo assumete che la stringa debba essere troncata. Non è necessario mettere il carattere null alla fine.
- (d) Copiare il contenuto di una variabile vettore all'interno di un'altra.
7. La Sezione 23.6 spiega come chiamare ripetutamente la funzione `strchr` in modo da individuare tutte le occorrenze di un carattere all'interno di una stringa. Chiamando ripetutamente la funzione `strrchr` è possibile individuare tutte le occorrenze in *ordine inverso*?
- W 8. Utilizzate la funzione `strchr` per scrivere la seguente funzione:
- ```
int numchar(const char *s, char ch);
```
- La funzione deve restituire il numero di occorrenze del carattere ch nella stringa s.
9. Sostituite la condizione di controllo della seguente istruzione if con una singola chiamata alla funzione `strchr`:
- ```
if (ch == 'a' || ch == 'b' || ch == 'c') ...
```
- W 10. Sostituite la condizione di controllo della seguente istruzione if con una singola chiamata alla `strstr`:
- ```
if (strcmp(str, "foo") == 0 || strcmp(str, "bar") == 0 || strcmp(str, "baz") == 0) ...
```
- Suggerimento:* combinate i letterali stringa in una singola stringa, separandoli con uno speciale carattere. La vostra soluzione fa delle assunzioni sul contenuto di str?
- W 11. Scrivete una chiamata alla funzione `memset` che sostituisca con dei caratteri ! gli ultimi n caratteri della stringa s (stringa terminante con il carattere null).
12. Molte versioni di `<string.h>` forniscono delle funzioni aggiuntive non standard come quelle elencate qui di seguito. Scrivete ognuna di queste funzioni usando solamente le possibilità offerte dal C standard.
- `strup(s)` – Restituisce un puntatore a una copia di s contenuta in una porzione di memoria ottenuta chiamando la funzione `malloc`. Restituisce un puntatore nullo nel caso non fosse possibile allocare memoria sufficiente.
  - `stricmp(s1, s2)` – Simile alla `strcmp` ma non tiene conto del fatto che le lettere siano maiuscole o minuscole.
  - `strlwr(s)` – Converte le lettere maiuscole presenti in s nella loro versione minuscola, lasciando inalterati gli altri caratteri. Restituisce s.
  - `strrev(s)` – Inverte l'ordine dei caratteri della stringa s (ad eccezione del carattere null). Restituisce s.
  - `strset(s, ch)` – Riempie s con delle copie del carattere ch. Restituisce s.

Se vorrete testare una qualsiasi di queste funzioni avrete bisogno di modificare il suo nome. Le funzioni i cui nomi iniziano per str sono riservate allo standard C.

13. Usate la funzione strtok per scrivere la funzione seguente:

```
int count_words(char *sentence);
```

Questa funzione restituisce il numero di parole contenute nella stringa sentence, dove per "parola" si intende una qualsiasi sequenza di caratteri che non rappresentano degli spazi bianchi. Alla funzione è permesso modificare la stringa.

## Progetti di programmazione

1. Scrivete un programma che cerchi le radici dell'equazione  $ax^2 + bx + c = 0$  usando la formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Fate in modo che il programma chieda all'utente i valori di  $a$ ,  $b$  e  $c$ , e che poi stampi entrambi i valori di  $x$  (nel caso il valore di  $b^2 - 4ac$  fosse negativo, il programma dovrà stampare un messaggio per avvisare che le radici sono complesse).

- W 2. Scrivete un programma che copi un file di testo dallo standard input allo standard output, rimuovendo tutti i caratteri di spazio bianco presenti all'inizio di ogni riga. Una riga che consistesse di soli spazi bianchi non dovrà essere copiata.
3. Scrivete un programma che copi un file di testo dallo standard input allo standard output facendo diventare maiuscola la prima lettera di ogni parola.
4. Scrivete un programma che chieda all'utente di inserire una serie di parole separate da un singolo spazio e poi stampi le parole in ordine inverso. Leggete l'input come una stringa e poi usate la funzione strtok per suddividerla in parole.
5. Supponete che del denaro venga depositato in un conto di risparmio e lì lasciato per  $t$  anni. Assumete che il tasso di interesse annuale sia pari a  $r$  e che questo interesse venga composto continuativamente. La formula  $A(t) = Pe^{rt}$  può essere usata per calcolare il valore finale del conto, dove  $P$  è l'ammontare depositato originalmente. Per esempio, 1000 \$ lasciati in deposito per 10 anni con un interesse del 6% verrebbero a valere  $1000 \times e^{0.06 \times 10} = 1000 \times e^0.6 = 1000 \times 1,8221188 = 1.822,12 \$$ . Scrivete un programma che visualizzi il risultato di questo calcolo dopo aver chiesto all'utente di immettere l'ammontare depositato in principio, il tasso di interesse e il numero di anni.
6. Scrivete un programma che copi un file di testo dallo standard input allo standard output, sostituendo ogni carattere di controllo (diverso da \n) con un punto di domanda.
7. Scrivete un programma che conti il numero di frasi presenti in un testo (ottenuto dallo standard input). Assumete che ogni frase termini con un ., ?, o ! seguito da un carattere di spazio bianco (\n incluso).



# 24 Gestione degli errori

Sebbene solitamente i programmi degli studenti non funzionino quando sono soggetti a input inaspettato, i programmi commerciali devono essere "invulnerabili" e, invece di andare in crash, devono essere in grado di riprendersi con grazia dagli eventuali errori. Per rendere i programmi a prova di proiettile ci viene richiesto di anticipare gli errori che possono sorgere durante l'esecuzione, di includere un controllo per ognuno di questi e di fornire al programma un'azione adeguata da eseguire nel caso se ne verificasse uno.

Questo capitolo descrive due modi per controllare gli errori: usare la macro `assert` e controllare la variabile `errno`. La Sezione 24.1 tratta l'header `<assert.h>`, dove la macro `assert` viene definita. La Sezione 24.2 discute l'header `<errno>` al quale appartiene la variabile `errno`. Questa sezione include anche una trattazione delle funzioni  `perror` e `strerror`. Queste funzioni, che derivano rispettivamente dagli header `<stdio.h>` e `<string.h>`, sono strettamente collegate alla variabile `errno`.

La Sezione 24.3 spiega come i programmi possono rilevare e gestire le condizioni conosciute come segnali, alcune delle quali rappresentano degli errori. Le funzioni che gestiscono i segnali vengono dichiarate nell'header `<signal.h>`. La Sezione 24.4, infine, esplora il meccanismo `setjmp/longjmp`, il quale viene spesso utilizzato in risposta agli errori. Sia `setjmp` che `longjmp` appartengono all'header `<setjmp.h>`.

Il rilevamento e la gestione degli errori non sono tra i punti di forza del C. Il C indica gli errori di *run-time* in modi diversi invece che in modo singolo e uniforme. Inoltre è responsabilità del programmatore includere del codice per controllare gli errori. È facile lasciarsi sfuggire possibili errori, e se dovessero verificarsi, spesso il programma continuerà a funzionare, anche se non molto bene. I nuovi linguaggi come C++, Java e C# possiedono una caratteristica di "gestione delle eccezioni" che facilita l'individuazione degli errori e la reazione verso questi.

## 24.1 L'header `<assert.h>`: diagnostica

```
void assert(scalar expression);
```

assert

La macro `assert`, che è definita nell'header `<assert.h>`, permette a un programma di monitorare il suo comportamento e rilevare i possibili problemi a uno stadio iniziale.

Sebbene assert sia effettivamente una macro, è progettata per essere utilizzata come una funzione. Ha un argomento che deve essere costituito da una "asserzione" (un'espressione che, in circostanze normali, ci aspettiamo sia vera). Ogni volta che la assert viene eseguita, controlla il valore del suo argomento. Se l'argomento ha un valore diverso da zero, la assert non fa nulla. Se il valore dell'argomento è pari a zero, la macro scrive un messaggio su stderr [stream stderr > 22.1] (lo standard error stream) e chiama la funzione abort [funzione abort > 26.2] per terminare l'esecuzione del programma.

Per esempio, supponiamo che il file demo.c dichiari il vettore a di lunghezza 10:

Siamo preoccupati che l'istruzione

```
a[i] = 0;
```

presente in demo.c possa causare dei problemi al programma per il fatto che il valore di i deve essere compreso tra 0 e 9. Possiamo utilizzare assert e controllare questa condizione prima di effettuare l'assegnamento ad a[i]:

```
assert(0 <= i && i < 10); /* prima controlla l'indice */
a[i] = 0; /* e poi effettua l'assegnamento */
```

Se il valore di i è minore di 0 oppure maggiore o uguale a 10, il programma terminerà dopo aver visualizzato un messaggio come il seguente:

```
Assertion failed: 0 <= i && i < 10, file demo.c, line 109
```



Il C99 ha effettuato un paio di modifiche minori alla macro assert. Lo standard C89 asserisce che l'argomento della assert debba essere di tipo int. Lo standard C99 attenua questo prerequisito permettendo che l'argomento sia di un qualsiasi tipo scalare (da qui la parola *scalar* presente nel prototipo della assert). Questa modifica, per esempio, permette che l'argomento sia un numero a virgola mobile o un punto fluttuante. Inoltre il C99 richiede che una assert non andata a buon fine visualizzi il nome della funzione nella quale compare (il C89 richiede solo che la assert visualizzi il nome dell'argomento, in forma testuale, assieme al nome del file sorgente e del numero della linea). La forma suggerita per il messaggio è:

```
Assertion failed: expression, function abc, file xyz, line nnn.
```

La forma esatta del messaggio prodotto dalla macro assert può variare da un compilatore all'altro, nonostante ciò deve sempre contenere le informazioni richieste dal standard. Il compilatore GCC, per esempio, nella situazione precedente produce il seguente messaggio:

```
a.out: demo.c:109: main: Assertion `0 <= i && i < 10' failed.
```

La macro assert presenta uno svantaggio: incrementa leggermente il tempo di esecuzione del programma a causa dei controlli extra che esegue. Utilizzare la assert una volta ogni tanto probabilmente non avrà un grande effetto sulla velocità del programma, tuttavia questa piccola penalità potrebbe essere inaccettabile in applicazioni critiche. La conseguenza è che molti programmati utilizzano la assert in fase di testing e poi la disabilitano quando il programma è terminato. Disabilitare la macro assert è facile, dobbiamo definire la macro NDEBUG prima dell'inclusione dell'header <assert.h>:

```
#define NDEBUG
#include <assert.h>
```

Il valore di `NDEBUG` non ha importanza, è importante solo che questa sia definita. S un secondo momento il programma dovesse incontrare problemi, potremmo sen riattivare la macro `assert` rimuovendo la definizione di `NDEBUG`.



Evitate di mettere all'interno di un `assert` un'espressione che abbia un side effect (incl una chiamata a funzione). Se in un secondo momento la macro `assert` venisse disabilit l'espressione non verrebbe più calcolata. Considerate l'esempio seguente:

```
assert((p = malloc(n)) != NULL);
```

Se `NDEBUG` venisse definita, la macro `assert` verrebbe ignorata e la funzione `malloc` n verrebbe chiamata.

## 24.2 L'header <errno.h>: errori

Alcune funzioni della libreria standard indicano i malfunzionamenti salvando un codice di errore (un intero positivo) all'interno di `errno`, una variabile di tipo `int` che viene dichiarata all'interno dell'header `<errno.h>` (errno, effettivamente, potrebbe essere una macro. In tal caso lo standard C richiede che rappresenti un lvalue [lvalue > 4.2], permettendoci di usarla come una variabile). La maggior parte delle funzioni che si basano su `errno` appartengono a `<math.h>`, ma ci sono alcune che appartengono ad altre parti della libreria.

Supponiamo di aver bisogno di usare una funzione di libreria che segnali un errore salvando un valore in `errno`. Dopo aver chiamato questa funzione possiamo controllare se il valore di `errno` è diverso da zero. In tal caso vorrebbe dire che si è verificato un errore durante la chiamata alla funzione. Supponete per esempio di voler controllare se una chiamata alla funzione `sqrt` (radice quadrata) [funzione `sqrt` > 23.3] non sia andata a buon fine. Ecco come si presenterebbe il codice:

```
errno = 0;
y = sqrt(x);
if (errno != 0) {
 fprintf(stderr, "sqrt error; program terminated.\n");
 exit(EXIT_FAILURE);
}
```



Quando `errno` viene usata per rilevare un errore in una chiamata a una funzione di libreria, è importante salvare uno zero al suo interno prima dell'invocazione della funzione. Sebbene il valore di `errno` sia uguale a zero all'inizio dell'esecuzione del programma, potrebbe essere alterato da una successiva chiamata ad una funzione. Le funzioni di libreria non azzerano mai la variabile `errno`, questa è responsabilità del programma.



Spesso il valore contenuto in `errno` a seguito di un errore, è `EDOM` o `ERANGE` (entrambe le macro sono definite in `<errno.h>`). Queste macro rappresentano due tipi di errori che possono verificarsi quando una funzione matematica viene chiamata:

- **Errori di dominio (EDOM).** Un argomento passato a una funzione è al di fuori del dominio della funzione. Per esempio, passare un numero negativo alla `sqrt` provoca un errore di dominio.
- **Errori di intervallo (ERANGE).** Il valore restituito da una funzione è troppo grande per essere rappresentato con il tipo restituito dalla funzione. Per esempio, di solito passare il valore 1000 alla funzione `exp` [funzione exp > 23.3] causa un errore di intervallo perché sulla maggior parte dei computer  $e^{1000}$  è troppo grande per essere rappresentato con un `double`.

Alcune funzioni possono essere soggette a entrambi gli errori, confrontando la variabile `errno` con `EDOM` o `ERANGE` possiamo determinare quale errore si sia verificato.

C99

Il C99 aggiunge a `<errno.h>` la macro `EILSEQ`. Le funzioni di libreria presenti in certi header (specialmente in `<wchar.h>`) [header <wchar> 25.5]) salvano il valore `EILSEQ` in `errno` quando si verificano errori di codifica [errori di codifica > 22.3].

## Le funzioni perror e strerror

```
void perror(const char *s); da <stdio.h>
char *strerror(int errnum); da <string.h>
```

Ci concentriamo ora su due funzioni che sono collegate alla variabile `errno` sebbene nessuna delle due appartenga ad `<errno.h>`.

**perror** Quando una funzione di libreria salva un valore diverso da zero nella variabile `errno` potremmo voler memorizzare un messaggio che indichi la natura dell'errore. Un modo per farlo è chiamare la funzione `perror` (dichiarata in `<stdio.h>`), la quale stampa nell'ordine i seguenti oggetti: (1) il suo argomento, (2) il carattere dei due punti, (3) uno spazio, (4) un messaggio di errore determinato dal valore di `errno` e (5) il carattere new-line. La funzione `perror` scrive sullo stream `stderr` [stream stderr > 22.1] e non nello standard output.

Ecco come potremmo usare la funzione `perror`:

```
errno = 0;
y = sqrt(x);
if (errno != 0) {
 perror("sqrt error");
 exit(EXIT_FAILURE);
}
```

Se la chiamata alla `sqrt` fallisce a causa di un errore di dominio, la `perror` genera il seguente output:

```
sqrt error: Numerical argument out of domain
```

Il messaggio di errore che la funzione `perror` visualizza dopo aver scritto `sqrt error` è definito dall'implementazione. In questo esempio, `Numerical argument out of domain` è il messaggio corrispondente all'errore `EDOM`. Un errore `ERANGE` di solito produce un messaggio diverso come: `Numerical result out of range`.

**strerror** La funzione strerror appartiene all'header <string.h>. Quando le viene passato un codice di errore, questa funzione restituisce un puntatore a una stringa contenente la descrizione dell'errore. Per esempio, la chiamata

```
puts(strerror(EDOM));
dovrebbe stampare il messaggio
Numerical argument out of domain
```

Di solito l'argomento della funzione strerror è uno dei valori posseduti da errno, tuttavia la funzione restituisce una stringa per qualsiasi intero le venga passato.

La strerror è strettamente collegata alla funzione perror. Il messaggio di errore che viene visualizzato dalla funzione perror è lo stesso che verrebbe restituito dalla strerror se le venisse passata la variabile errno come argomento.

## 24.3 L'header <signal.h>: gestione dei segnali

L'header <signal.h> fornisce mezzi per la gestione delle condizioni eccezionali conosciute come **segnali**. I segnali si dividono in due categorie: errori di run-time (come una divisione per zero) ed eventi causati al di fuori del programma. Molti sistemi operativi, per esempio, permettono agli utenti di interrompere o "uccidere" i programmi che sono in esecuzione. Questi eventi, nel C, vengono trattati come segnali. Quando si verifica un errore o un evento esterno, diciamo che un segnale è stato **generato**. Molti segnali sono asincroni: possono accadere in qualsiasi momento durante l'esecuzione del programma e non solo in certi punti noti al programmatore. Dato che i segnali possono presentarsi in momenti inaspettati, devono essere trattati in un modo unico.

Questa sezione tratta i segnali così come vengono descritti dallo standard C. I segnali giocano in UNIX un ruolo più importante di quello che ci si può aspettare dalla trattazione limitata che viene presentata qui. Per maggiori informazioni sui segnali UNIX consultate uno dei libri di programmazione UNIX che sono citati nella bibliografia.

### Macro per i segnali

**D&R** L'header <signal.h> definisce un certo numero di macro che rappresentano i **segnali**. La Tabella 24.1 elenca queste macro e il loro significato. Il valore di ogni macro è un intero positivo costante. Alle implementazioni del C è permesso fornire altre macro per i segnali, ammesso che i loro nomi inizino per SIG seguito da una lettera maiuscola (le implementazioni UNIX, in particolare, forniscono un gran numero di macro aggiuntive).

Lo standard C non richiede che i segnali della Tabella 24.1 vengano generati automaticamente dal momento che non tutti possono avere significato per un particolare computer e sistema operativo. La maggior parte delle implementazioni supporta almeno alcuni di questi segnali.

**Tabella 24.1** Segnali

| Nome    | Significato                                                                                             |
|---------|---------------------------------------------------------------------------------------------------------|
| SIGABRT | Interruzione anormale (forse causata da una chiamata alla funzione <code>abort</code> )                 |
| SIGFPE  | Errore durante un'operazione aritmetica (può essere causato da una divisione per zero o da un overflow) |
| SIGILL  | Istruzione non valida                                                                                   |
| SIGINT  | Interruzione                                                                                            |
| SIGSEGV | Accesso alla memoria non valido                                                                         |
| SIGTERM | Richiesta di interruzione                                                                               |

## La funzione signal

```
void (*signal(int sig, void (*func)(int)))(int);
```

**signal** L'header `<signal.h>` fornisce due funzioni: `raise` e `signal`. Inizieremo con la funzione `signal`, la quale installa una funzione per la gestione di un segnale in modo che possa essere utilizzata se in un secondo momento dovesse verificarsi il segnale stesso. L'uso di questa funzione è più facile di quello che ci si potrebbe aspettare dal suo quasi intimidatorio prototipo. Il suo primo argomento è il codice di un particolare segnale. Il secondo argomento è un puntatore a una funzione che gestirà il segnale nel caso in cui quest'ultimo venisse generato durante l'esecuzione del programma. Per esempio, la seguente chiamata alla funzione `signal` installa un *handler* (o gestore) per il segnale SIGINT:

```
signal(SIGINT, handler);
```

`handler` è il nome di una funzione per la gestione del segnale. Se il segnale SIGINT si verifica durante l'esecuzione del programma, la funzione `handler` verrà chiamata automaticamente.

Tutte le funzioni per la gestione dei segnali devono possedere un parametro `int` e restituire il tipo `void`. Quando un particolare segnale viene generato e viene chiamato il suo `handler`, a quest'ultimo viene passato il codice del segnale stesso. Conoscere quale segnale ha causato la sua chiamata può essere utile per l'`handler`, perché ci permette di utilizzare il medesimo gestore per diversi segnali.

Una funzione `handler` può effettuare diverse cose: può ignorare il segnale, eseguire qualche tipo di recupero dalla condizione di errore o terminare il programma. Tuttavia, a meno che non sia stato invocato dalla funzione `abort` [funzione `abort` > 26.2] o dalla funzione `raise`, il gestore per un segnale non deve invocare funzioni di libreria o cercare di utilizzare variabili con durata di memorizzazione statica [durata di memorizzazione statica > 18.2] (ci sono alcune eccezioni a queste regole).

Se un `handler` ha termine, il programma riprende la sua esecuzione dal punto nel quale il segnale l'aveva interrotto, a meno che non ci si trovi in uno di questi due casi: (1) il segnale era SIGABRT, allora il programma terminerà (in modo anormale) non appena l'`handler` ha termine; (2) l'effetto del ritorno da una funzione che ha gestito il segnale SIGFPE non è definito (in altre parole, non fatelo).

Sebbene la funzione signal abbia un valore restituito, spesso questo viene scartato. Se lo si desidera, il valore restituito, che è un puntatore all'handler precedente per lo specifico segnale, può essere salvato all'interno di una variabile. In particolare se pianifichiamo di ripristinare il gestore originario per segnale, abbiamo la necessità di salvare il valore restituito dalla funzione signal:

```
void (*orig_handler)(int); /* variabile puntatore a funzione */
-
orig_handler = signal(SIGINT, handler);
```

Questa istruzione installa handler come gestore di SIGINT e salva un puntatore al gestore originario nella variabile orig\_handler. Per ripristinare il gestore originario dobbiamo scrivere

```
signal(SIGINT, orig_handler); /* ripristina l'handler originario */
```

## Handler predefiniti per i segnali

Invece di scrivere i nostri handler per i segnali, possiamo usare uno di quelli predefiniti che vengono forniti dall'header <signal.h>. Ne sono presenti due, ognuno rappresentato da una macro:

- **SIG\_DFL.** SIG\_DFL gestisce i segnali in un modo di "default". Per installare SIG\_DFL possiamo usare una chiamata come questa:

```
signal(SIGINT, SIG_DFL); /* usa handler */
```

L'effetto di chiamare SIG\_DFL è definito dall'implementazione, ma nella maggior parte dei casi provoca il termine del programma.

- **SIG\_IGN.** La chiamata

```
signal(SIGINT, SIG_IGN); /* ignora il segnale SIGINT */
```

specificà che il segnale SIGINT deve essere ignorato.

In aggiunta a SIG\_DFL e SIG\_IGN, l'header <signal.h> può fornire degli altri handler per i segnali. I loro nomi devono iniziare per SIG\_ seguito da una lettera maiuscola. All'inizio dell'esecuzione del programma, l'handler per ogni segnale viene inizializzato a SIG\_DFL o a SIG\_IGN a seconda dell'implementazione.

L'header <signal.h> definisce un'altra macro, SIG\_ERR, la quale sembra essere un handler. In effetti questa macro viene utilizzata per testare il verificarsi di un eventuale errore durante l'installazione di un handler. Se una chiamata alla signal non va a buon fine (non può installare un handler per uno specifico segnale) restituisce il valore SIG\_ERR e salva un valore positivo nella variabile errno. Quindi, per controllare se la funzione signal non è andata a buon fine, possiamo scrivere

```
if (signal(SIGINT, handler) == SIG_ERR) {
 perror("signal(SIGINT, handler) failed");
}
```

C'è solo un aspetto delicato nell'intero meccanismo della gestione dei segnali: cosa succede se un segnale viene generato da una funzione che gestisce il segnale stesso? Per prevenire una ricorsione infinita, lo standard C89 prescrive un processo in due passi nel caso in cui viene generato un segnale per il quale il programmatore ha installato una funzione di gestione. Per prima cosa, o l'handler per il segnale viene reimpostato a SIG\_DFL (l'handler di default) oppure il segnale viene bloccato durante l'esecuzione dell'handler (SIGILL è un caso speciale, nessuna azione è richiesta quando il segnale SIGILL viene sollevato). Solo successivamente viene invocato l'handler fornito dal programmatore.



Dopo che un segnale è stato gestito, dipende dall'implementazione se l'handler debba essere reinstallato o meno. Tipicamente le implementazioni UNIX lasciano gli handler installati dopo il loro uso, tuttavia altre implementazioni possono reimpostare l'handler del segnale a SIG\_DFL. Nell'ultimo caso l'handler può reinstallare se stesso chiamando la funzione signal prima del suo termine.



Il C99 ha modificato il processo in alcuni modi minori. Quando un segnale viene generato, un'implementazione può scegliere di disabilitare non solo quel segnale ma anche gli altri. Se un handler termina dopo aver gestito i segnali SIGILL o SIGSEGV (o allo stesso modo il segnale SIGFPE), l'effetto non è definito. Il C99 aggiunge inoltre la restrizione secondo la quale se un segnale si verifica come risultato di una chiamata alla funzione abort o alla funzione raise, l'handler stesso non deve chiamare la funzione raise.

## La funzione raise

```
int raise(int sig);
```

raise

Sebbene di solito i segnali vengano generati da errori di run-time o da eventi esterni, a volte per un programma è comodo provocare il verificarsi di un dato segnale. La funzione raise fa esattamente questo. L'argomento di questa funzione specifica il codice per il segnale desiderato:

```
raise(SIGABRT); /* genera il segnale SIGABRT */
```

Il valore restituito dalla funzione raise può essere usato per controllare se la chiamata è andata a buon fine: lo zero indica un successo, mentre un valore diverso da zero indica un fallimento.

MINIMAMMA

## Testare i segnali

Il programma seguente illustra l'uso dei segnali. Per prima cosa, il programma installa un handler per il segnale SIGINT (salvando diligentemente l'handler originale) e poi chiama la funzione raise\_sig per generare il segnale. Successivamente il programma installa SIG\_INT come handler per il segnale SIGINT e chiama nuovamente la raise\_sig. Il programma, infine, reinstalla l'handler originale per SIG\_INT e poi chiama la funzione raise\_sig un'ultima volta.

```

tsignal.c /* Testa i segnali */

#include <signal.h>
#include <stdio.h>

void handler(int sig);
void raise_sig(void);

int main(void)

{
 void (*orig_handler)(int);
 printf("Installing handler for signal %d\n", SIGINT);
 orig_handler = signal(SIGINT, handler);
 raise_sig();

 printf("Changing handler to SIG_IGN\n");
 signal(SIGINT, SIG_IGN);
 raise_sig();

 printf("Restoring original handler\n");
 signal(SIGINT, orig_handler);
 raise_sig();

 printf("Program terminates normally\n");
 return 0;
}

void handler(int sig)
{
 printf("Handler called for signal %d\n", sig);
}

void raise_sig(void)
{
 raise(SIGINT);
}

```

Tra l'altro, la chiamata alla funzione `raise` non ha bisogno di trovarsi in una funzione separata. Abbiamo definito la funzione `raise_sig` semplicemente per chiarire un punto: indipendentemente da dove un segnale venga generato (se nella funzione `main` o in un'altra funzione), questo verrà gestito dall'handler che è stato installato più di recente per il segnale stesso.

L'output di questo programma può variare in qualche modo. Ecco una possibilità:

```

Installing handler for signal 2
Handler called for signal 2
Changing handler to SIGN_IGN
Restoring original handler

```

Da questo output possiamo vedere che la nostra implementazione definisce `SIGINT` come il valore 2 e che l'handler originale per `SIGINT` deve essere stato `SIG_DFL` (se

-fosse stato `SIG_IGN` avremmo visto anche il messaggio `Program terminates normally`. Possiamo osservare infine, che `SIG_DFL` ha provocato la fine del programma senza visualizzare un messaggio di errore.

## 24.4 L'header <setjmp.h>: salti non locali

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

Normalmente una funzione ritorna al punto nel quale è stata chiamata. Non possiamo usare un'istruzione `goto` [istruzione `goto` > 6.4] per farla andare altrove perché quest'ultima può saltare solamente a un'etichetta che si trovi all'interno della stessa funzione. L'header `<setjmp.h>` invece rende possibile per una funzione di saltare direttamente a un'altra funzione senza effettuare il `return`.

Gli oggetti più importanti presenti in `<setjmp.h>` sono la macro `setjmp` e la funzione `longjmp`. La `setjmp` "segna" una posizione all'interno di un programma, la funzione `longjmp` può essere usata per ritornare in quel punto in un secondo momento. Sebbene questo meccanismo potente si presti a diverse applicazioni, viene utilizzato principalmente per la gestione degli errori.

`setjmp`



`longjmp`

Per segnare un obiettivo di un futuro salto, chiamiamo la `setjmp`, passandole una variabile di tipo `jmp_buf` (dichiarata in `<setjmp.h>`). La macro `setjmp` salva l'ambiente corrente (che include un puntatore alla locazione della stessa `setjmp`) nella variabile per un uso successivo in una chiamata alla `longjmp` e poi restituisce il valore zero. Ritornare al punto dove è stata invocata la `setjmp` viene fatto per mezzo di un'invocazione alla `longjmp`, passandole la stessa variabile `jmp_buf` che era stata passata alla prima delle due. Dopo aver ripristinato l'ambiente rappresentato dalla variabile `jmp_buf`, la `longjmp` (qui è dove la cosa si fa complicata) *ritornerà dalla chiamata alla setjmp*. Questa volta il valore restituito dalla `setjmp` è `val`, il secondo argomento della `longjmp` (se `val` è uguale a 0 la `setjmp` restituisce 1).



Assicuratevi che gli argomenti alla `longjmp` vengano prima inizializzati da una chiamata alla `setjmp`. È importante anche che la funzione contenente la chiamata originale alla `setjmp` non debba mai terminare prima della chiamata alla `longjmp`. Se una di queste restrizioni viene violata, una chiamata alla `longjmp` risulterà in un comportamento indefinito (probabilmente il programma andrà in crash).

Riassumendo: la `setjmp` restituisce uno zero la prima volta che viene chiamata, successivamente la `longjmp` trasferisce nuovamente il controllo alla chiamata originaria alla `setjmp`, la quale, questa volta, restituisce un valore diverso da zero. Forse è bene guardare un esempio.

PROGRAMMA

### Testare `setjmp/longjmp`

Il programma seguente utilizza la macro `setjmp` per segnare una posizione all'interno del `main`. La funzione `f2` successivamente ritorna in quella posizione chiamando la funzione `longjmp`.

```
tsetjmp.c /* Testa setjmp/longjmp */

#include <setjmp.h>
#include <stdio.h>

jmp_buf env;

void f1(void);
void f2(void);

int main(void)
{
 if (setjmp(env) == 0)
 printf("setjmp returned 0\n");
 else {
 printf("Program terminates: longjmp called\n");
 return 0;
 }

 f1();
 printf("Program terminates normally\n");
 return 0;
}

void f1(void)
{
 printf("f1 begins\n");
 f2();
 printf("f1 returns\n");
}

void f2(void)
{
 printf("f2 begins\n");
 longjmp(env, 1);
 printf("f2 returns\n");
}
```

L'output di questo programma sarà

```
setjmp returned 0
f1 begins
f2 begins
Program terminates: longjmp called
```

La chiamata originale alla `setjmp` restituisce 0 e quindi il `main` chiama la funzione `f1`. Successivamente la `f1` chiama la `f2`, che utilizza la `longjmp` per ritrasferire il controllo al `main` invece di ritornare alla `f1`. Quando la `longjmp` viene eseguita, il controllo ritornerà alla chiamata della `setjmp`. Questa volta la `setjmp` restituisce il valore 1 (ovvero il valore specificato nella chiamata alla `longjmp`).

## Domande & Risposte

**D:** Lei ha detto che è importante salvare il valore zero in `errno` prima di chiamare una funzione di libreria che potrebbe modificarla. Tuttavia abbiamo visto dei programmi UNIX che controllano il valore di `errno` senza mai impostarla al valore zero. Come si spiega tutto questo? [p. 651]

**R:** Spesso i programmi UNIX contengono chiamate a funzioni che appartengono al sistema operativo. Queste chiamate di sistema si affidano alla variabile `errno`, tuttavia la utilizzano in un modo leggermente diverso da quanto descritto in questo capitolo. Quando una chiamata del genere non va a buon fine, restituisce uno speciale valore (come -1 o un puntatore nullo) oltre che a salvare un valore nella variabile `errno`. I programmi non hanno bisogno di salvare uno zero in `errno` prima di una chiamata di questo tipo perché il valore restituito dalla funzione indica da solo che si è verificato un errore. Anche alcune funzioni dello standard C funzionano in questo modo, non usando `errno` tanto per segnalare l'errore quanto per specificare che errore fosse.

**D:** La nostra versione di `<errno.h>` definisce altre macro oltre alla `EDOM` e alla `ERANGE`. Questa pratica è ammisible? [p. 651]

**R:** Sì. Lo standard C ammette delle macro che rappresentano altre condizioni di errore, ammesso che i loro nomi inizino con la lettera E seguita da una cifra o da una lettera maiuscola. Le implementazioni UNIX tipicamente definiscono un numero enorme di questo tipo di macro.

**D:** Alcune delle macro che rappresentano i segnali possiedono nomi criptici, come `SIGFPE` e `SIGSEGV`. Da dove derivano questi nomi? [p. 653]

**R:** Questi nomi risalgono ai primi compilatori C che venivano eseguiti su un DEC PDP-11. L'hardware del PDP-11 è in grado di rilevare errori con nomi come "Floating Point Exception" e "Segmentation Violation".

**D:** A meno che non sia stato invocato dalle funzioni `abort` o `raise`, l'handler per un segnale non deve chiamare funzioni appartenenti alla libreria standard. Tuttavia vi sono delle eccezioni, quali sono? [p. 654]

**R:** All'handler di un segnale è permesso chiamare la funzione `signal`, ammesso che il primo argomento corrisponda al segnale che sta gestendo al momento. Questa condizione è importante perché permette all'handler di un segnale di reinstallarsi. Nel C99 l'handler di un segnale può chiamare anche la funzione `abort` o la funzione `_Exit` [funzione `_Exit` > 26.2].

**\*D:** Seguendo la domanda precedente, normalmente l'handler per un segnale non dovrebbe accedere a variabili con durata della memorizzazione statica. Qual è l'eccezione a questa regola?

**R:** È una domanda difficile. La risposta coinvolge un tipo chiamato `sig_atomic_t` che è dichiarato nell'header `<signal.h>`. `sig_atomic_t` è un tipo intero al quale, secondo lo standard C, si può accedere come "entità atomica" (*atomic entity*). In altre parole la CPU può caricare dalla memoria o salvare nella memoria un valore `sig_atomic_t` con una singola istruzione macchina invece di doverne usare due o più. Spesso questo tipo

è definito uguale al tipo int perché la maggior parte delle CPU può caricare o salvare un valore int in una sola istruzione.

Questo ci porta all'eccezione alla regola che l'handler di un segnale non debba accedere a variabili statiche. Lo standard C permette a un handler di salvare un valore in una variabile `sig_atomic_t` (anche con durata di memorizzazione statica) ammesso che questa sia dichiarata volatile [**qualificatore volatile > 20.3**]. Per capire la ragione di questa regola arcana considerate cosa potrebbe accadere se l'handler di un segnale dovesse modificare una variabile statica che fosse di un tipo più grande di `sig_atomic_t`. Se il programma avesse caricato dalla memoria una parte della variabile un attimo prima che il segnale venisse generato e terminasse il caricamento dopo che il segnale sia stato gestito, si ritroverebbe con un dato "spazzatura". Le variabili `sig_atomic_t` possono essere caricate in un singolo colpo, di conseguenza il problema non si verifica. Dichiarare una variabile come volatile avverte il compilatore che il valore della variabile può modificarsi in ogni momento (un segnale può essere generato di lì a poco, invocando un handler che modifica la variabile).

**D: Il programma `tsignal.c` chiama la funzione `printf` dall'interno di un handler per il segnale. Questo non era proibito?**

**R:** Una funzione handler per un segnale invocata come risultato della funzione `raise` o della `abort` può chiamare delle funzioni di libreria. Il programma `tsignal.c` utilizza la funzione `raise` per invocare l'handler del segnale.

**D: Come fa `setjmp` a modificare un argomento che non le viene passato? Pensavamo che il C passasse gli argomenti sempre per valore. [p. 658]**

**R:** Lo standard C dice che `jmp_buf` deve essere di un tipo vettore, quindi alla `setjmp` viene effettivamente passato un puntatore.

**D: Stiamo incontrando dei problemi con la funzione `setjmp`. Ci sono alcune restrizioni sul suo utilizzo?**

**R:** Secondo lo standard C, sono ammessi solamente due modi di utilizzare la `setjmp`:

- come un'espressione in un expression statement (eventualmente con un cast al tipo `void`);
- come parte di un'espressione di controllo in un'istruzione `if`, `switch`, `while`, `do` o `for`. L'intera espressione di controllo deve avere uno dei seguenti formati, dove `constespr` è un'espressione costante di tipo intero e `op` è un operatore relazionale o di uguaglianza:

```
setjmp(...)
!setjmp(...)
constespr op setjmp(...)
setjmp(...) op constespr
```

Usare la `setjmp` in qualsiasi altro modo provoca un comportamento indefinito.

**D: Dopo che un programma ha eseguito una chiamata alla `longjmp`, quali sono i valori delle variabili?**

**R:** La maggior parte delle variabili manterrà il valore che avevano al momento della `longjmp`. Tuttavia una variabile automatica all'interno di una funzione che contenga la

`setjmp` possiede un valore non determinato a meno che non sia stata dichiarata volatile o che non sia stata modificata dal momento in cui era stata eseguita la `setjmp`.

**D:** È possibile chiamare la `longjmp` all'interno dell'handler per un segnale?

**R:** Sì, ammesso che l'handler non fosse stato invocato a causa di un segnale generato durante l'esecuzione dell'handler di un segnale (il C99 rimuove questa restrizione).

C99

## Esercizi

### Sezione 24.1

1. (a) La macro `assert` può essere usata per due tipi di problemi: (1) i problemi che non dovrebbero mai verificarsi se il programma fosse corretto; (2) i problemi che vanno al di là del controllo del programma. Spiegate perché la macro `assert` è particolarmente indicata per i problemi della prima categoria.
- (b) Fornite tre esempi di problemi che vanno al di là del controllo del programma.
2. Scrivete una chiamata alla `assert` che faccia terminare il programma nel caso in cui una variabile chiamata `top` ha valore `NULL`.
3. Modificate il file `stackADT2.c` della Sezione 19.4 in modo che per testare gli errori utilizzi la macro `assert` invece di usare l'istruzione `if` (osservate che la funzione `terminate` non è più necessaria e può essere rimossa).

### Sezione 24.2

W

4. (a) Scrivete una funzione "wrapper" chiamata `try_math_fcn` che chiama una funzione matematica (assumete che abbia un argomento di tipo `double` e che il valore restituito sia di tipo `double`) e poi controlli se la chiamata è andata a buon fine. Ecco come dovremmo usare la `try_math_fcn`:

```
y = try_math_fcn(sqrt, x, "Error in call of sqrt");
```

Se la chiamata `sqrt(x)` ha avuto successo, la `try_math_fcn` restituisce il valore calcolato dalla `sqrt`. Se la chiamata non è andata a buon fine, la `try_math_fcn` chiama la  `perror` per stampare il messaggio `Error in call of sqrt` e poi chiama la funzione `exit` per terminare il programma.

- (b) Scrivete una macro che abbia lo stesso effetto della funzione `try_math_fcn` ma che costruisca il messaggio di errore a partire dal nome della funzione:

```
y = TRY_MATH_FCN(sqrt, x);
```

Se la chiamata alla `sqrt` non va a buon fine, il messaggio sarà `Error in call of sqrt`. *Suggerimento:* fate in modo che `TRY_MATH_FCN` chiama la funzione `try_math_fcn`.

### Sezione 24.4

W

5. Nel programma `inventory.c` (Sezione 16.3) la funzione `main` possiede un ciclo `for` che chiede all'utente di immettere un codice operativo, legge tale codice e poi chiama una delle funzioni `insert`, `search`, `update` o `print`. Aggiungete nel `main` una chiamata alla `setjmp` in modo che una successiva chiamata alla `longjmp` ritorni al ciclo `for` (dopo la `longjmp` all'utente verrà chiesto un codice operativo e il programma continuerà normalmente). La `setjmp` avrà bisogno della variabile `jmp_buf`, dove dovrà essere dichiarata?

# 25 Internazionalizzazione

Per molti anni l'uso del C non è stato particolarmente adatto per i Paesi non anglofoni. Originariamente il C assumeva che i caratteri fossero sempre singoli byte e che tutti i computer riconoscessero i caratteri #, [ \ ], ^, { , | , } e ~, necessari per scrivere i programmi. Sfortunatamente queste assunzioni non sono valide in tutte le parti del mondo. Di conseguenza gli esperti che crearono il C89 aggiunsero al linguaggio caratteristiche e librerie nello sforzo di rendere il C un linguaggio più internazionale.

Nel 1994 è stato approvato l'Amendment 1 (Revisione 1) dello standard ISO C, creando una versione potenziata del C89 che a volte viene chiamata C94 o C95. Questa modifica fornisce un supporto aggiuntivo della libreria per la programmazione internazionale attraverso le funzionalità digrafiche e gli header <iso646.h>, <wchar.h> e <wctype.h>. Il C99 ha aggiunto un supporto ancora maggiore per l'internazionalizzazione sotto forma degli *universal character name*. Questo capitolo tratta tutte le caratteristiche di internazionalizzazione del C, sia quelle del C89 che quelle dell'Amendment 1 e del C99. Le modifiche dell'Amendment 1 verranno segnalate come modifiche del C99 anche se in effetti sono precedenti a quest'ultimo.

L'header <locale.h> (Sezione 25.1) fornisce delle funzioni che permettono a un programma di adattare il suo comportamento a una particolare localizzazione (spesso una nazione o un'area geografica nella quale viene parlata una particolare lingua). I caratteri multibyte e i wide character (Sezione 25.2) permettono ai programmi di lavorare con un grande set di caratteri come quello usato nelle nazioni asiatiche. Digrafi, trigrafi e l'header <iso646.h> (Sezione 25.3) rendono possibile la scrittura di programmi su computer che non possiedono alcuni dei caratteri normalmente usati nella programmazione C. Gli universal character name (Sezione 25.4) permettono ai programmati di incorporare nel codice di un programma i caratteri tratti dall'Universal Character Set. L'header <wchar.h> (Sezione 25.5) fornisce funzioni per l'input/output dei wide character e per la manipolazione di stringhe formate da wide character. Infine l'header <wctype.h> (Sezione 25.6) fornisce delle funzioni per la classificazione e il case-mapping dei wide character.

## 25.1 L'header <locale.h>: localizzazione

L'header <locale.h> fornisce delle funzioni per controllare porzioni della libreria C il cui comportamento varia da una localizzazione all'altra (tipicamente una **localizzazione** o *locale* è una nazione o una regione nella quale viene parlata una particolare lingua).

Gli aspetti della libreria dipendenti dalla localizzazione includono:

- **formattazione delle quantità numeriche.** Per esempio in alcune localizzazioni il separatore decimale è il punto (297.48) mentre in altre è la virgola (297,48);
- **formattazione delle quantità monetarie.** Per esempio il simbolo della valuta cambia da nazione a nazione.
- **set di caratteri.** Spesso il set di caratteri dipende dalla lingua di una particolare localizzazione. Le nazioni Asiatiche solitamente richiedono un set di caratteri più esteso rispetto alle nazioni occidentali;
- **aspetto della data e dell'ora.** In alcune localizzazioni, nella scrittura di una data è abitudine mettere prima il mese (8/24/2012), mentre in altre viene messo prima il giorno (24/8/2012).

### Categorie

Modificando la localizzazione, un programma può adattare il suo comportamento a differenti aree del mondo. Tuttavia il cambio di localizzazione può riguardare molte parti della libreria, alcune delle quali potremmo preferire che non venissero alterate. Fortunatamente non ci viene richiesto di cambiare tutti gli aspetti di una localizzazione allo stesso tempo. Possiamo invece utilizzare una delle seguenti macro per specificare una **categoria**.

- **LC\_COLLATE.** Riguarda il comportamento delle funzioni di confronto tra due stringhe `strcoll` e `strxfrm` (entrambe le funzioni sono dichiarate in <string.h> [header <string.h> > 23.6]).
- **LC\_CTYPE.** Riguarda il comportamento delle funzioni presenti in <cctype.h> [header <cctype.h> > 23.5] (a eccezione di `isdigit` e `isxdigit`). Interessa anche le funzioni per i multibyte e i wide character discusse in questo capitolo.
- **LC\_MONETARY.** Riguarda la formattazione delle informazioni monetarie restituite dalla funzione `localeconv`.
- **LC\_NUMERIC.** Riguarda il carattere del separatore decimale usato dalle funzioni di I/O formattato (come la `printf` e la `scanf`) e dalle funzioni di conversione numerica [funzioni di conversione numerica > 26.2] (come la `strtod`) presenti in <stdlib.h>. Interessa anche la formattazione delle informazioni non monetarie restituite da `localeconv`.
- **LC\_TIME.** Riguarda il comportamento della funzione `strftime` [funzione strftime > 26.3] (dichiarata in <time.h>), la quale converte un orario in una stringa di caratteri. Nel C99 interessa anche il comportamento della funzione `wcsftime` [funzione wcsftime > 25.5].

Le varie implementazioni sono libere di fornire delle categorie aggiuntive e definire delle macro LC\_ non elencate qui sopra. Per esempio, la maggior parte dei sistemi UNIX fornisce una categoria LC\_MESSAGES, la quale riguarda il formato dei responsi affermativi e negativi.

## La funzione setlocale

```
char *setlocale(int category, const char *locale);
```

### setlocale

La funzione setlocale cambia la localizzazione corrente, sia per una singola categoria che per tutte le categorie. Se il primo argomento è una delle macro LC\_COLLATE, LC\_CTYPE, LC\_MONETARY, LC\_NUMERIC o LC\_TIME la chiamata alla set locale interesserà una sola categoria. Se il primo argomento è uguale a LC\_ALL la chiamata riguarderà tutte le categorie. Lo standard C definisce solo due valori per il secondo argomento: "C" e "" . Altre localizzazioni, se presenti, dipendono dall'implementazione.

All'inizio dell'esecuzione del programma, viene eseguita la chiamata

```
setlocale(LC_ALL, "C");
```

Nella localizzazione "C", le funzioni di libreria si comportano nel modo "normale" e il separatore decimale corrisponde al punto.

Modificare la localizzazione dopo che il programma ha iniziato la sua esecuzione richiede una chiamata esplicita alla funzione setlocale. Chiamare la setlocale con "" come suo secondo argomento passa alla **localizzazione nativa**, permettendo così al programma di adattare il suo comportamento all'ambiente del luogo. Lo standard C non definisce l'esatto effetto del passare alla localizzazione nativa. Alcune implementazioni della funzione setlocale controllano l'ambiente di esecuzione (allo stesso modo della funzione getenv [[funzione getenv > 26.2](#)]), alla ricerca di una variabile d'ambiente con un nome particolare (per esempio lo stesso di una macro di categoria). Altre implementazioni non fanno nulla (lo standard C non richiede che la funzione setlocale abbia qualche effetto. Naturalmente una libreria la cui versione di setlocale non fa nulla, probabilmente non verrà venduta molto in alcune parti del mondo).

## Localizzazioni

Le localizzazioni diverse da "C" e "" cambiano da un compilatore all'altro. La libreria GNU C, conosciuta come glibc, fornisce una localizzazione "POSIX", che è uguale alla localizzazione "C". La libreria glibc, che viene utilizzata da Linux, permette che vengano installate delle localizzazioni aggiuntive se lo si desidera. Queste localizzazioni hanno il formato:

```
language [_territory] [codeset] [@modifier]
```

dove ogni oggetto racchiuso tra parentesi quadre è opzionale. Possibili valori per l'oggetto *language* sono elencati in uno standard conosciuto come ISO 639, l'opzione *territory* deriva da un altro standard (ISO 3166) e il *codeset* specifica un set di caratteri o la codifica di un set di caratteri. A pagina seguente alcuni esempi.

"swedish" (Svedese)  
 "en\_GB" (Inglese - Regno Unito)  
 "en\_IE" (Inglese - Irlanda)  
 "fr\_CH" (Francese - Svizzera)

C'erano diverse varianti della localizzazione "en\_IE", incluse la "en\_IE@euro" (che utilizza l'euro come valuta), la "en\_IE.iso88591" (che utilizza il set di caratteri ISO/IEC 8859-1), la "en\_IE.iso88591@euro" (che utilizza il set di caratteri ISO/IEC 8859-15 e l'euro) e la "en\_IE.utf8" (che utilizza la codifica UTF-8 del set di caratteri Unicode).

Linux e altre versioni di UNIX supportano il comando `locale`, che può essere utilizzato per ottenere delle informazioni sulla localizzazione. Uno degli utilizzi del comando `locale` è quello di ottenere una lista di tutte le localizzazioni disponibili scrivendo

`locale -a`

sulla riga di comando.

A causa del fatto che le informazioni di localizzazione diventano sempre più importanti, il Consorzio Unicode ha creato il progetto *Common Locale Data Repository* (CLDR) per stabilire un insieme di localizzazioni standard. Maggiori informazioni sul progetto CLDR possono essere trovate al link [www.unicode.org/cldr/](http://www.unicode.org/cldr/).

Quando una chiamata alla `setlocale` ha successo, questa restituisce un puntatore a una stringa associata con la categoria nella nuova localizzazione (per esempio la stringa può essere il nome della localizzazione stessa). In caso di insuccesso la funzione restituisce un puntatore nullo.

La funzione `setlocale` può essere usata anche come una funzione di interrogazione. Se il suo secondo argomento è un puntatore nullo, la funzione restituisce un puntatore a una stringa associata con la categoria presente nella localizzazione corrente. Questa caratteristica è particolarmente utile se il primo argomento è `LC_ALL`, visto che ci permette di ottenere le impostazioni correnti per tutte le categorie. Una stringa restituita dalla funzione `setlocale` può essere salvata (copiandola in una variabile) e poi utilizzata in una successiva chiamata alla stessa funzione.



## La funzione `localeconv`

```
struct lconv *localeconv(void);
```

`localeconv` Sebbene possiamo chiedere alla funzione `setlocale` delle informazioni sulla localizzazione corrente, queste non vengono necessariamente restituite nella forma più utile. Per trovare delle informazioni particolarmente specifiche a riguardo della localizzazione corrente (qual è il carattere per il separatore decimale? Qual è il simbolo per la valuta?) abbiamo bisogno della funzione `localeconv`, ovvero l'unica altra funzione dichiarata in `<locale.h>`.

La funzione `localeconv` restituisce un puntatore a una struttura del tipo `struct lconv`. I membri di questa struttura contengono informazioni dettagliate sulla localizzazione corrente. La struttura ha una durata di memorizzazione statica e può essere modificata da una successiva chiamata alle funzioni `localeconv` o `setlocale`. Assicu-

ratevi di estrarre le informazioni desiderate dalla struttura `lconv` prima che venga modificata da una di queste funzioni.

Alcuni dei membri della struttura `lconv` sono di tipo `char *`, gli altri sono di tipo `char`. La Tabella 25.1 elenca i membri `char *`. I primi tre membri descrivono la formattazione delle quantità non monetarie, mentre gli altri hanno a che fare con le quantità monetarie. La tabella illustra anche il valore di ogni membro nella localizzazione "C" (quella di default), il valore "" significa "non disponibile".

**Tabella 25.1** I membri `char *` della struttura `lconv`

|               | Nome              | Valore nella localizzazione "C" | Descrizione                                                                 |
|---------------|-------------------|---------------------------------|-----------------------------------------------------------------------------|
| Non monetarie | decimal_point     | ":"                             | Carattere del separatore decimale.                                          |
|               | thousands_sep     | " "                             | Carattere usato per separare gruppi di cifre prima del separatore decimale. |
|               | grouping          | ""                              | Dimensioni dei gruppi di cifre.                                             |
| Monetarie     | mon_decimal_point | ""                              | Carattere del separatore decimale.                                          |
|               | mon_thousands_sep | ""                              | Carattere usato per separare gruppi di cifre prima del separatore decimale. |
|               | mon_grouping      | ""                              | Dimensioni dei gruppi di cifre.                                             |
|               | positive_sign     | ""                              | Stringa indicante quantità non negativa.                                    |
|               | negative_sign     | ""                              | Stringa indicante quantità negativa.                                        |
|               | currency_symbol   | ""                              | Simbolo della valuta locale.                                                |
|               | int_curr_symbol   | ""                              | Simbolo della valuta internazionale. <sup>1</sup>                           |

<sup>1</sup>Un'abbreviazione di tre lettere seguite da un separatore (spesso uno spazio o un punto). Per esempio, i simboli internazionali delle valute della Svizzera, del Regno Unito e degli Stati Uniti sono rispettivamente "(CH) " "GBP" e "USD".

I membri `grouping` e `mon_grouping` meritano una menzione speciale. Ogni carattere in queste stringhe specifica la dimensione di un gruppo di cifre (il raggruppamento avviene da destra a sinistra iniziando dal separatore decimale). Un valore `CHAR_MAX` indica che non deve essere eseguito nessun ulteriore raggruppamento. Lo 0 indica che l'elemento precedente deve essere usato per le cifre rimanenti. Per esempio, la stringa "\3" (\3 seguito da \0) indica che il primo gruppo deve essere di 3 cifre e poi tutte le altre cifre devono a loro volta essere raggruppate a gruppi di 3.

I membri `char` della struttura `lconv` sono divisi in due gruppi. I membri del primo gruppo (Tabella 25.2) interessano la formattazione locale delle quantità monetarie. I membri del secondo gruppo (Tabella 25.3) hanno a che fare con la formattazione internazionale delle quantità monetarie. Tutti i membri della Tabella 25.3 eccetto uno sono stati aggiunti dal C99. Come mostrano le Tabelle 25.2 e 25.3, il valore di ogni membro `char` presente nella localizzazione "C" è pari a `CHAR_MAX`, che significa "non disponibile".

**Tabella 25.2** I membri char della struttura lconv (formattazione locale)

| Nome           | Valore nella<br>localizzazione "C" | Descrizione                                                                                               |
|----------------|------------------------------------|-----------------------------------------------------------------------------------------------------------|
| frac_digits    | CHAR_MAX                           | Numero di cifre dopo il separatore decimale.                                                              |
| p_cs_precedes  | CHAR_MAX                           | 1 se currency_symbol precede una quantità non negativa, 0 se segue la quantità.                           |
| n_cs_precedes  | CHAR_MAX                           | 1 se currency_symbol precede una quantità negativa, 0 se segue la quantità.                               |
| p_sep_by_space | CHAR_MAX                           | La separazione di currency_symbol e la stringa di segno da una quantità non negativa (vedi Tabella 25.4). |
| n_sep_by_space | CHAR_MAX                           | La separazione di currency_symbol e la stringa di segno da una quantità negativa (vedi Tabella 25.4).     |
| p_sign_posn    | CHAR_MAX                           | Posizione di positive_sign per una quantità non negativa (vedi Tabella 25.5).                             |
| n_sign_posn    | CHAR_MAX                           | Posizione di negative_sign per una quantità negativa (vedi Tabella 25.5).                                 |

**Tabella 25.3** I membri char della struttura lconv (formattazione internazionale)

| Nome                            | Valore nella<br>localizzazione "C" | Descrizione                                                                                               |
|---------------------------------|------------------------------------|-----------------------------------------------------------------------------------------------------------|
| int_frac_digits                 | CHAR_MAX                           | Numero di cifre dopo il separatore decimale.                                                              |
| int_p_cs_precedes <sup>t</sup>  | CHAR_MAX                           | 1 se int_curr_symbol precede una quantità non negativa, 0 se segue la quantità.                           |
| int_n_cs_precedes <sup>t</sup>  | CHAR_MAX                           | 1 se int_curr_symbol precede una quantità negativa, 0 se segue la quantità.                               |
| int_p_sep_by_space <sup>t</sup> | CHAR_MAX                           | La separazione di int_curr_symbol e la stringa di segno da una quantità non negativa (vedi Tabella 25.4). |
| int_n_sep_by_space <sup>t</sup> | CHAR_MAX                           | La separazione di int_curr_symbol e la stringa di segno da una quantità negativa (vedi Tabella 25.4).     |
| int_p_sign_posn <sup>t</sup>    | CHAR_MAX                           | Posizione di positive_sign per una quantità non negativa (vedi Tabella 25.5).                             |
| int_n_sign_posn <sup>t</sup>    | CHAR_MAX                           | Posizione di negative_sign per una quantità negativa (vedi Tabella 25.5).                                 |

<sup>t</sup>solo C99

C99

La Tabella 25.4 spiega il significato dei valori dei membri `p_sep_by_space`, `n_sep_by_space`, `int_p_sep_by_space` e `int_n_sep_by_space`. Il significato di `p_sep_by_space` e `n_sep_by_space` è cambiato nel C99. Nel C89 c'erano solamente due possibili valori per questi membri: 1 (se c'è uno spazio tra `currency_symbol` e la quantità monetaria) e 0 (se lo spazio non è presente).

**Tabella 25.4** Valori dei membri ...`sep_by_space`

| Valore | Descrizione                                                                                                                                                                       |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0      | Nessuno spazio separa il simbolo della valuta dalla quantità.                                                                                                                     |
| 1      | Se il simbolo della valuta e quello del segno sono adiacenti, uno spazio li separa dalla quantità, altrimenti uno spazio separa il simbolo della valuta da quello della quantità. |
| 2      | Se il simbolo della valuta e quello del segno sono adiacenti, uno spazio li separa, altrimenti uno spazio separa il segno dalla quantità.                                         |

La Tabella 25.5 spiega il significato dei valori dei membri `p_sign_posn`, `n_sign_posn`, `int_p_sign_posn` e `int_n_sign_posn`.

**Tabella 25.5** Valori dei membri ...`sign_posn`

| Valore | Descrizione                                                       |
|--------|-------------------------------------------------------------------|
| 0      | Delle parentesi circondano la quantità e il simbolo della valuta. |
| 1      | Il segno precede la quantità e il simbolo della valuta.           |
| 2      | Il segno segue la quantità e il simbolo della valuta.             |
| 3      | Il segno precede immediatamente il simbolo della valuta.          |
| 4      | Il segno segue immediatamente il simbolo della valuta.            |

Per capire come i membri della struttura `lconv` possano variare da una localizzazione all'altra guardiamo due esempi. La Tabella 25.6 illustra i valori tipici dei membri monetari di `lconv` per gli U.S.A. e la Finlandia (che utilizza l'euro come valuta).

**Tabella 25.6** Valori tipici dei membri monetari di `lconv` per gli U.S.A. e la Finlandia

| Membro                         | U.S.A. | Finlandia |
|--------------------------------|--------|-----------|
| <code>mon_decimal_point</code> | "."    | ","       |
| <code>mon_thousands_sep</code> | ";"    | " "       |
| <code>mon_grouping</code>      | "\3"   | "\3"      |
| <code>positive_sign</code>     | "+"    | "+"       |
| <code>negative_sign</code>     | "_"    | "_"       |
| <code>currency_symbol</code>   | "\$"   | "EUR"     |
| <code>frac_digits</code>       | 2      | 2         |
| <code>p_cs_precedes</code>     | 1      | 0         |
| <code>n_cs_precedes</code>     | 1      | 0         |
| <code>p_sep_by_space</code>    | 0      | 2         |
| <code>n_sep_by_space</code>    | 0      | 2         |

| Membro             | U.S.A. | Finlandia |
|--------------------|--------|-----------|
| p_sign_posn        | 1      | 1         |
| n_sign_posn        | 1      | 1         |
| int_curr_symbol    | "USD " | "EUR "    |
| int_frac_digits    | 2      | 2         |
| int_p_cs_precedes  | 1      | 0         |
| int_n_cs_precedes  | 1      | 0         |
| int_p_sep_by_space | 1      | 2         |
| int_n_sep_by_space | 1      | 2         |
| int_p_sign_posn    | 1      | 1         |
| int_n_sign_posn    | 1      | 1         |

Ecco come la quantità monetaria 7593.86 verrebbe formattata nelle due localizzazioni a seconda del suo segno e del fatto che la formattazione sia locale o internazionale:

|                                   | U.S.A.        | Finlandia      |
|-----------------------------------|---------------|----------------|
| Formato locale (positivo)         | \$7,593.86    | 7 593,86 EUR   |
| Formato locale (negativo)         | -\$7,593.86   | - 7 593,86 EUR |
| Formato internazionale (positivo) | USD 7,593.86  | 7 593,86 EUR   |
| Formato internazionale (negativo) | -USD 7,593.86 | - 7 593,86 EUR |

Tenete presente che nessuna delle funzioni di libreria del C è in grado di formattare automaticamente delle quantità monetarie. Dipende dal programmatore l'utilizzo delle informazioni presenti nella struttura lconv per realizzare la formattazione.

## 25.2 Caratteri multibyte e wide character

Uno dei più grandi problemi nell'adattare i programmi alle diverse localizzazioni è il set di caratteri. Il set di caratteri ASCII e le sue estensioni, le quali includono il set Latin-1 [[Latin-1 > 7.3](#)], sono i più diffusi nel Nord America. Altrove la situazione è più complicata. In molti Paesi i computer impiegano dei set di caratteri che sono simili all'ASCII ma sono privi di certe caratteristiche. Discuteremo di questo nella Sezione 25.3. Altri Paesi, specialmente in Asia, devono affrontare un problema diverso: i linguaggi scritti richiedono un set di caratteri molto ampio, tipicamente nell'ordine delle migliaia di caratteri.

Modificare il significato del tipo char al fine di gestire un set di caratteri più estesi non è possibile, dato che i valori char sono (per definizione) limitati ai singoli byte. Invece il C permette ai compilatori di fornire un **set di caratteri estesi**. Questo set di caratteri può essere usato per scrivere dei programmi C (nei commenti e nelle stringhe per esempio), negli ambienti nei quali il programma viene eseguito o in entrambe le situazioni. Il C prevede due tecniche per la codifica di un set di caratteri esteso: i caratteri multibyte e i wide character. Il C fornisce anche delle funzioni che convertono da un tipo di codifica all'altro.

## Caratteri multibyte

In una codifica a **carattere multibyte**, ogni carattere esteso viene rappresentato da una sequenza di uno o più byte. Il numero di byte può variare a seconda del carattere. Il C richiede che ogni set di caratteri esteso debba includere certi caratteri essenziali (lettere, cifre, operatori, punteggiatura e caratteri di spazio bianco). Questi caratteri devono essere singoli byte. Gli altri byte possono essere interpretati come l'inizio di un carattere multibyte.

## Set di caratteri giapponese

I giapponesi impiegano diversi sistemi di scrittura. Il più complesso, il *kanji*, consiste di migliaia di simboli, decisamente troppi per poterli rappresentare con una codifica a un byte (i simboli *kanji* derivano dal cinese, che possiede un problema simile con i grandi set di caratteri). Non c'è un singolo modo per codificare il *kanji*, le codifiche più comuni includono la JIS (*Japanese Industrial Standard*), la Shift-JIS (la codifica più popolare) e la EUC (*Extended UNIX Code*).

Alcuni set di caratteri multibyte si basano su una **codifica dipendente dallo stato**. In questo tipo di codifica, ogni sequenza di caratteri multibyte inizia con uno **stato di shift iniziale** (*initial shift state*). Certi byte incontrati successivamente (conosciuti come **sequenza di shift** o *shift sequence*) possono modificare lo stato dello shift, influendo sul significato dei byte seguenti. La codifica giapponese JIS, per esempio, mischia codici a un byte con codici a due byte. Le "sequenze di escape" incorporate nelle stringhe indicano quando passare dalla modalità a un byte a quella a due byte e viceversa (la codifica Shift-JIS al contrario non è dipendente dallo stato. Ogni carattere richiede uno o due byte. Il primo byte di un carattere a due byte è sempre distinguibile da un carattere costituito da un byte singolo).

In qualsiasi codifica lo standard C richiede che un byte a zero rappresenti sempre il carattere null e questo indipendentemente dallo stato di shift. Inoltre il secondo (o successivo) byte di un carattere multibyte non può essere uguale a zero.

La libreria C fornisce due macro, la `MB_LEN_MAX` e la `MB_CUR_MAX`, che sono **collegate ai caratteri multibyte**. Entrambe le macro specificano il numero massimo di byte presenti in un carattere multibyte. La macro `MB_LEN_MAX` (definita in `<limits.h>`) dà il numero massimo supportato da qualsiasi localizzazione, la `MB_CUR_MAX` (definita in `<stdlib.h>`) fornisce il numero massimo per la localizzazione corrente (modificare la localizzazione può avere effetti sull'interpretazione dei caratteri multibyte). Ovviamente `MB_CUR_MAX` non può essere maggiore di `MB_LEN_MAX`.

Una qualsiasi stringa può contenere dei caratteri multibyte, sebbene la sua lunghezza (come determinato dalla funzione `strlen`) corrisponda al numero di byte presenti e non al numero di caratteri. In particolare le stringhe di formato nelle chiamate delle funzioni `...printf` e `...scanf` possono contenere dei caratteri multibyte. Come risultato lo standard C99 definisce il termine **stringa multibyte** come sinonimo di *stringa*.

## Wide character

L'altro modo per codificare un set di caratteri esteso è quello di usare i wide character. Un **wide character** è un intero il cui valore rappresenta un carattere. A differenza dei caratteri multibyte, i quali possono variare in lunghezza, tutti i wide character supportati da una particolare implementazione richiedono lo stesso numero di byte. Una **stringa wide** è una stringa che consiste di wide character che ha un wide character null alla fine (un **wide character null** è un wide character il cui valore numerico è pari a zero).

I wide character sono di tipo `wchar_t` (dichiarato in `<stddef.h>` e in alcuni altri header), il quale deve essere un tipo intero in grado di rappresentare il più grande set di caratteri esteso di ogni localizzazione supportata. Per esempio, se due byte sono sufficienti per rappresentare un qualsiasi set di caratteri esteso, allora `wchar_t` può essere definito come `unsigned short int`.

Il C supporta sia le costanti wide character che le stringhe letterali costituite da wide character. Le costanti wide character somigliano alle costanti carattere ordinarie ma iniziano con la lettera `L`:

`L'a'`

Anche le stringhe letterali costituite da wide character sono precedute dal prefisso `L`:

`L"abc"`

Questa stringa rappresenta un vettore contenente i wide character `L'a'`, `L'b'` e `L'c'` seguiti da un wide character null.

## Unicode e l'Universal Character Set

Le differenze tra i caratteri multibyte e i wide character diventano visibili quando si parla di **Unicode**. Unicode è un enorme set di caratteri sviluppato dall'*Unicode Consortium*, un'organizzazione fondata da un gruppo di produttori di computer per creare un set di caratteri internazionale. I primi 256 caratteri del set Unicode sono identici a quelli del set Latin-1 (quindi i primi 128 caratteri del set Unicode combaciano con il set di caratteri ASCII). Tuttavia l'Unicode si spinge ben oltre il Latin-1 fornendo i caratteri necessari per quasi tutti i linguaggi moderni e antichi. Questo set include anche un certo numero di simboli specializzati, come quelli usati in matematica e per la musica. Lo standard Unicode è stato pubblicato per la prima volta nel 1991.

Unicode è strettamente collegato con lo standard internazionale ISO/IEC 10646, il quale definisce una codifica di caratteri conosciuta come **Universal Character Set (UCS)**. L'UCS è stato sviluppato dall'*International Organization for Standardization (ISO)*, iniziando più o meno nello stesso periodo nel quale è stato inizialmente definito lo standard Unicode. Sebbene originariamente l'UCS fosse diverso da Unicode, i due set di caratteri sono stati successivamente unificati. Attualmente l'ISO lavora a stretto contatto con l'Unicode Consortium al fine di assicurare che lo standard ISO/IEC 10646 rimanga consistente con l'Unicode. A causa del fatto che i due standard sono così simili useremo i termini Unicode e USC in modo intercambiabile.

Originariamente l'Unicode era limitato a 65,536 caratteri (il numero di caratteri che possono essere rappresentati usando 16 bit). Successivamente questo limite si

rivelò insufficiente. Attualmente l'Unicode possiede più di 100,000 caratteri (per la versione più recente visitate [www.unicode.org](http://www.unicode.org)). I primi 65.536 caratteri dello standard (che includono i caratteri usati più frequentemente) sono conosciuti come **Basic Multilingual Plane (BMP)**.

## Codifiche per Unicode

Lo standard Unicode assegna a ogni carattere un numero univoco conosciuto come **code point**. Ci sono diversi modi per rappresentare questi code point usando dei byte, noi vedremo solo due delle tecniche più semplici. Una di queste codifiche utilizza i wide character mentre l'altra usa i caratteri multibyte.

La **UCS-2** è una codifica a wide character nella quale ogni code point viene rappresentato come una coppia di byte. La UCS-2 può rappresentare tutti i caratteri del *Basic Multilingual Plane* (ovvero tutti i code point compresi in esadecimale tra 0000 e FFFF) ma non è in grado di rappresentare i caratteri Unicode che non appartengono al BMP.

Un'alternativa piuttosto diffusa è costituita dall'**UTF-8** (*8-bit UCS Transformation Format*), il quale utilizza i caratteri multibyte. L'UTF-8 venne pensato nel 1992 da Ken Thompson e il suo collega Rob Pike nei Laboratori Bell (lo stesso Ken Thompson che ha sviluppato il linguaggio B, il predecessore del C). L'UTF-8 possiede l'utile proprietà che fa sì che in questa codifica i caratteri ASCII appaiano identici: ogni carattere è costituito da un byte e ha la stessa codifica binaria. Quindi il software sviluppato per leggere i dati UTF-8 può gestire anche i dati ASCII senza bisogno di modifiche. Per questa ragione l'UTF-8 è largamente utilizzato nelle applicazioni Internet basate sul testo come le pagine web e le e-mail.

In UTF-8 ogni code point richiede da uno a quattro byte. L'UTF-8 è organizzato in modo che, come si può vedere dalla Tabella 25.7, i caratteri più usati richiedano meno byte.

**Tabella 25.7** Codifica UTF-8

| Intervallo dei code point (esadecimale) | Sequenza di Byte UTF-8 (binario)    |
|-----------------------------------------|-------------------------------------|
| 000000-00007F                           | 0xxxxxx                             |
| 000080-0007FF                           | 110xxxxx 10xxxxxx                   |
| 000800-00FFFF                           | 1110xxxx 10xxxxxx 10xxxxxx          |
| 010000-10FFFF                           | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx |

L'UTF-8 prende i bit del valore del code point e li divide in gruppi (rappresentati dalle x nella Tabella 25.7) e assegna a ogni gruppo un byte differente. Il caso più semplice è un code point nell'intervallo 0-7F (un carattere ASCII), il quale viene rappresentato da uno 0 seguito dai sette bit del numero originale.

Un code point appartenente all'intervallo 80-7FF (che include tutti i caratteri Latin-1) vede i suoi bit suddivisi in due gruppi di cinque e sei bit. Il gruppo da cinque bit è preceduto dal prefisso 110 mentre il gruppo a sei bit è preceduto dal prefisso 10. Per esempio, il code point del carattere à è E4 (esadecimale) o 11100100

(binario). Tale carattere verrebbe rappresentato nell'UTF-8 da una sequenza di due byte 11000011 10100100. Osservate come le parti sottolineate, quando unite assieme, formino il numero 00011100100.

I caratteri i cui code point ricadono all'interno del range 800–FFFF, ovvero quelli che includono i rimanenti caratteri del Basic Multilingual Plane, richiedono tre byte. A tutti gli altri caratteri Unicode (la maggior parte dei quali viene usata raramente) sono associati quattro byte.

Il numero UTF-8 possiede alcune proprietà utili:

- ognuno dei 128 caratteri ASCII è rappresentato da un solo byte. Una stringa consistente di soli caratteri ASCII si presenta esattamente uguale nella codifica UTF-8;
- ogni byte presente in una stringa UTF-8 il cui bit più significativo è pari a 0 deve essere un carattere ASCII perché tutti gli altri byte iniziano con un bit a 1;
- il primo byte di un carattere multibyte indica quanto sarà lungo il carattere stesso. Se il numero di bit a 1 all'inizio del byte è pari a 2, allora il carattere è lungo due byte. Se il numero di bit a 1 è pari a tre o quattro, il carattere è lungo rispettivamente tre o quattro byte;
- ogni altro byte in una sequenza multibyte ha i bit più significativi pari a 10.

Le ultime tre proprietà sono particolarmente importanti perché garantiscono che nessuna sequenza di byte all'interno di un carattere multibyte possa rappresentare un altro carattere multibyte valido. Questo rende possibile la ricerca all'interno di una stringa multibyte di un particolare carattere o di una sequenza di caratteri, semplicemente attuando un confronto tra i byte.

Quindi come si pone l'UTF-8 nei confronti dell'UCS-2? L'UCS-2 possiede il vantaggio di memorizzare i caratteri nella loro forma più naturale. Dal canto suo, l'UTF-8 può gestire tutti i caratteri Unicode (e non solo quelli del BMP), spesso richiede meno spazio rispetto all'UCS-2 e mantiene la compatibilità con il set ASCII. L'UCS-2 non è diffuso quanto l'UTF-8 sebbene sia stato usato nel sistema operativo Windows NT. Una nuova versione che utilizza quattro byte (UCS-4) sta gradualmente prendendo il suo posto. Alcuni sistemi estendono l'UCS-2 in una codifica multibyte permettendo a un numero variabile di coppie di byte di rappresentare un carattere (diversamente dall'UCS-2, il quale utilizza una singola coppia di byte per ogni carattere). Questa codifica, conosciuta come **UTF-16** possiede il vantaggio di essere compatibile con l'UCS-2.

## Funzioni di conversione tra caratteri multibyte e wide character

```
int mblen(const char *s, size_t n); da <stdlib.h>
int mbtowc(wchar_t * restrict pwc,
 const char * restrict s,
 size_t n); da <stdlib.h>
int wctomb(char *s, wchar_t wc); da <stdlib.h>
```

Sebbene lo standard C89 introduca il concetto di caratteri multibyte e wide character, fornisce solo cinque funzioni per lavorare con queste tipologie di caratteri. Ora descriveremo queste funzioni, le quali appartengono all'header <stdlib.h>. Gli header <wchar.h> e <wctype.h> del C99, che sono discussi nelle Sezioni 25.5 e 25.6, forniscono un certo numero di funzioni aggiuntive per i caratteri multibyte e i wide character.

Le funzioni del C89 per i caratteri multibyte e i wide character sono divise in due gruppi. Il primo gruppo converte i singoli caratteri dalla forma multibyte alla forma wide character e viceversa. Il comportamento di queste funzioni dipende dalla categoria LC\_CTYPE della localizzazione corrente. Se la codifica multibyte è dipendente dallo stato, il comportamento dipende a sua volta dal corrente **stato di conversione** (*conversion state*). Lo stato di conversione consiste sia del corrente stato di shift che della corrente posizione all'interno di un carattere multibyte. Chiamare una di queste funzioni con un puntatore nullo come valore del suo parametro `char *` impone il suo stato di conversione interno allo **stato di conversione iniziale** (*initial conversion state*), il che significa che nessun carattere multibyte è già in analisi e che è attivo lo shift state iniziale. Chiamate successive della funzione fanno sì che lo stato di conversione interno venga aggiornato.

**mblen** La funzione `mblen` controlla se il suo primo argomento punta a una serie di byte che formano un carattere multibyte valido. In tal caso, la funzione restituisce un puntatore al carattere null, altrimenti restituisce -1. Come caso speciale questa funzione restituisce uno zero se il primo argomento punta al carattere null. Il secondo argomento limita il numero di byte che dovranno essere esaminati dalla funzione, tipicamente `MB_CUR_MAX`.

La funzione seguente, che proviene dal libro *The Standard C Library* di P.J. Plauger, utilizza la funzione `mblen` per determinare se una stringa consiste di caratteri multibyte validi. La funzione restituisce uno zero se `s` punta a una stringa valida.

```
int mbcheck(const char *s)
{
 int n;

 for (mblen(NULL, 0); ; s += n)
 if ((n = mblen(s, MB_CUR_MAX)) <= 0)
 return n;
}
```

Due aspetti della funzione `mbcheck` meritano una considerazione particolare. Per prima cosa c'è la misteriosa chiamata `mblen(NULL, 0)` che imposta lo stato di conversione interna della `mblen` allo stato di conversione iniziale. Secondariamente c'è la questione del termine. Tenete presente che `s` punta a una normale stringa di caratteri, che si assume termini con il carattere null. La funzione `mblen` restituisce uno zero quando raggiunge questo carattere null, provocando il termine della funzione `mbcheck`. La funzione `mbcheck` termina prima se la `mblen` restituisce un -1 a causa di un carattere multibyte non valido.

**mbtowc** La funzione `mbtowc` converte un carattere multibyte (puntato dal secondo argomento) in un wide character. Il primo argomento punta a una variabile `wchar_t` nella quale la funzione dovrà porre il risultato. Il terzo argomento pone un limite al numero di

byte che la `mbtowc` andrà a esaminare. La `mbtowc` restituisce lo stesso valore di `mblen`: il numero di byte presenti nel carattere multibyte se questo è valido, -1 se non lo è, zero se il secondo argomento punta al carattere null.

**wctomb**

La funzione `wctomb` converte un wide character (il secondo argomento) in un carattere multibyte, il quale viene salvato nel vettore puntato dal primo argomento. La `wctomb` può salvare nel vettore fino a `MB_LEN_MAX` caratteri, ma non aggiunge il carattere null. Questa funzione restituisce il numero di byte presenti nel carattere multibyte, oppure -1 nel caso in cui il wide character non corrispondesse a nessun carattere multibyte valido (osservate che la `wctomb` restituisce un 1 se le viene chiesto di convertire il wide character null).

La funzione seguente (anch'essa tratta dal libro *The Standard C Library* di Plauger) utilizza la `wctomb` per determinare se una stringa di wide character può essere convertita in caratteri multibyte validi:

```
int wccheck(wchar_t *wcs)
{
 char buf[MB_LEN_MAX];
 int n;

 for (wctomb(NULL, 0); ; ++wcs)
 if ((n = wctomb(buf, *wcs)) <= 0)
 return -1; /* carattere non valido */
 else if (buf[n-1] == '\0')
 return 0; /* tutti i caratteri sono validi */
}
```

Tra l'altro tutte e tre le funzioni (`mblen`, `mbtowc` e `wctomb`) possono essere utilizzate per controllare se una codifica multibyte è dipendente dallo stato. Quando le viene passato un puntatore nullo al posto del suo argomento `char *`, ognuna di queste funzioni restituisce un valore diverso da zero se i caratteri multibyte possiedono una codifica dipendente dallo stato oppure zero se ne sono privi.

## Funzioni di conversione tra stringhe multibyte e stringhe wide character

|                                                       |                                             |                         |                            |
|-------------------------------------------------------|---------------------------------------------|-------------------------|----------------------------|
| <code>size_t mbstowcs(wchar_t * restrict pwcs,</code> | <code>const char * restrict s,</code>       | <code>size_t n);</code> | <i>da &lt;stdlib.h&gt;</i> |
| <code>size_t wcstombs(char * restrict ps,</code>      | <code>const wchar_t * restrict pwcs,</code> | <code>size_t n);</code> | <i>da &lt;stdlib.h&gt;</i> |

Le rimanenti funzioni C89 per i caratteri multibyte e i wide character si occupano di convertire una stringa contenente dei caratteri multibyte in una stringa di wide character e viceversa. Come la conversione venga eseguita dipende dalla categoria `LC_CTYPE` della localizzazione corrente.

- mbstowcs** La funzione `mbstowcs` converte una sequenza di caratteri multibyte in una sequenza di wide character. Il secondo argomento punta a un vettore contenente il carattere multibyte che deve essere convertito. Il primo argomento punta a un vettore di wide character. Il terzo argomento pone un limite al numero di wide character che possono essere messi nel vettore. Questa funzione si ferma quando raggiunge il limite o incontra un carattere null (il quale viene salvato nel vettore dei wide character). La `mbstowcs` restituisce il numero di elementi del vettore che sono stati modificati, non includendo il wide character null, se presente. La funzione restituisce il valore -1 (al quale viene applicato un cast al tipo `size_t`) se incontra un carattere multibyte non valido.
- wcstombs** La funzione `wcstombs` è l'opposto della `mbstowcs`: converte una sequenza di wide character in una sequenza di caratteri multibyte. Il secondo argomento punta alla stringa di wide character. Il primo argomento punta al vettore nel quale i caratteri multibyte devono essere salvati. Il terzo argomento pone un limite al numero di byte che possono essere salvati nel vettore. La `wcstombs` si ferma quando raggiunge il limite o incontra un carattere null (il quale viene salvato). Questa funzione restituisce il numero di byte salvati senza includere il carattere null di termine (se presente). La funzione restituisce il valore -1 (con un cast al tipo `size_t`) se incontra un wide character che non corrisponde a nessun carattere multibyte.

La funzione `mbstowcs` assume che la stringa che deve essere convertita inizi con lo stato di shift iniziale. La stringa creata da `wcstombs` inizia sempre con lo stato di shift iniziale.

## 25.3 Digrafi e trigrafi

Tradizionalmente in certi Paesi i programmati hanno riscontrato problemi nello scrivere programmi C perché la loro tastiera è sprovvista di alcuni caratteri che sono richiesti dal C. Questo è particolarmente vero in Europa, dove le tastiere più vecchie fornivano i caratteri accentati usati nelle lingue Europee al posto dei caratteri necessari per il C come #, [ , ], ^, {, |, } e ~. Il C89 ha introdotto i trigrafi (dei codici di tre caratteri che rappresentano dei caratteri problematici) come soluzione a questo problema. I trigrafi, tuttavia, si dimostrarono impopolari e quindi l'Amendment 1 dello standard aggiunse due miglioramenti: i digrafi, che sono molto più leggibili dei trigrafi, e l'header `<iso646.h>`, il quale definisce delle macro che rappresentano certi operatori C.

### Trigrafi

Una sequenza trigrafica (o semplicemente, un trigrafo) è un codice di tre caratteri che può essere usato come alternativa a un carattere ASCII. La Tabella 25.8 fornisce la lista completa dei trigrafi. Tutti i trigrafi iniziano con ??, il che non li rende esattamente attrattivi, ma almeno sono facilmente individuabili.

**Tabella 25.8** Sequenze trigrafiche

| Sequenza trigrafica | Equivalenti ASCII |
|---------------------|-------------------|
| ??=                 | #                 |
| ??(                 | [                 |
| ??/                 | \                 |
| ??)                 | ]                 |
| ??'                 | ^                 |
| ??<                 | {                 |
| ??!                 |                   |
| ??>                 | }                 |
| ??-                 | ~                 |

I trigrafi possono essere liberamente sostituiti dai loro equivalenti ASCII. Per esempio, il programma

```
#include <stdio.h>

int main(void)
{
 printf("hello, world\n");
 return 0;
}
```

potrebbe essere scritto in questo modo:

```
??=include <stdio.h>

int main(void)
??<
 printf("hello, world??/n");
 return 0;
??>
```

I compilatori che sono conformi agli standard C89 e C99 devono accettare i trigrafi, anche se questi vengono usati raramente. A volte però questa caratteristica può causare dei problemi.



Fate attenzione a mettere la sequenza di caratteri ?? in una stringa letterale: è possibile che il compilatore la interpreti come l'inizio di un trigrafo. Se questo dovesse succedere, modificate il secondo carattere ? in una sequenza di escape facendolo precedere dal carattere \. La combinazione risultante ?\? non può essere scambiata per un trigrafo.

C99

## Digrafi

Riconoscendo che i trigrafi sono difficili da leggere, l'Amendment 1 dello standard C89 ha aggiunto una notazione alternativa conosciuta come **digrafi**. Come implica il nome, un digrafo è una sequenza di due soli caratteri al posto di tre. I digrafi sono disponibili come sostituti per i sei token [token > 2.8] illustrati nella Tabella 25.9.

**Tabella 25.9** Digrafi

| Digrafo | Token |
|---------|-------|
| <:      | [     |
| ::>     | ]     |
| <%      | {     |
| %>      | }     |
| %:      | #     |
| %::%    | ##    |

I digrafi, a differenza dei trigrafi, sono dei sostituti per i *token* e non dei sostituti per i *caratteri*. Di conseguenza i digrafi non verranno riconosciuti all'interno di una stringa letterale o di una costante carattere. Per esempio, la stringa "<::>" ha una lunghezza pari a quattro, contiene i caratteri: <, :, : e >, ma non i caratteri [ e ]. Per contro la stringa "??(??)" ha una lunghezza pari a due, perché il compilatore sostituisce il trigrafo ??( con il carattere [ e il trigrafo ??) con il carattere ].

I digrafi sono più limitati dei trigrafi. Per prima cosa, come abbiamo già visto, i digrafi non hanno alcuna utilità all'interno delle stringhe letterali o nelle costanti carattere. I trigrafi sono ancora necessari in queste situazioni. Secondariamente i digrafi non risolvono il problema di fornire una rappresentazione alternativa per i caratteri \, ^, | e ~. L'header <iso646.h> descritto più avanti, aiuta a risolvere questo problema.

## L'header <iso646.h>: grafie alternative

L'header <iso646.h> è piuttosto semplice. Non contiene altro che le definizioni delle undici macro che sono illustrate nella Tabella 25.10. Ognuna di queste macro rappresenta un operatore del C che contiene uno dei caratteri &, |, ~, ! e ^, rendendo così possibile l'uso degli operatori elencati nella tabella anche quando questi caratteri sono assenti nella tastiera.

**Tabella 25.10** Macro definite in <iso646.h>

| Macro  | Valore |
|--------|--------|
| and    | &&     |
| and_eq | &=     |
| bitand | &      |
| bitor  |        |
| compl  | ~      |
| not    | !      |
| not_eq | !=     |
| or     |        |
| or_eq  | =      |
| xor    | ^      |
| xor_eq | ^=     |

Il nome dell'header deriva da ISO/IEC 646, un vecchio standard per un set di caratteri simile all'ASCII. Questo standard permette delle "varianti nazionali", nelle quali i

diversi Paesi utilizzano dei caratteri locali al posto di certi caratteri ASCII, causando così il problema che i digrafi e l'header `<iso646.h>` stanno cercando di risolvere.

## 25.4 Universal Character Name (C99)

La Sezione 25.2 ha discusso dell'Universal Character Set (UCS), il quale è strettamente collegato con l'Unicode. Il C99 prevede una speciale caratteristica (**gli universal character name**) che ci permette di utilizzare i caratteri UCS nel codice sorgente di un programma.

Un universal character name somiglia a una sequenza di escape. Tuttavia a differenza delle normali sequenze di escape, che possono comparire solo nelle costanti carattere e nelle stringhe letterali, gli universal character name possono essere usati anche come identificatori. Questa caratteristica permette ai programmatore di utilizzare la loro lingua madre quando definiscono i nomi per le variabili, le funzioni e così via.

Ci sono due modi per scrivere un universal character name, `\udddd` e `\Uddddddd`, dove ogni *d* è una cifra esadecimale. Nella forma `\Uddddddd`, le *d* formano un numero esadecimale a otto cifre che identifica il code point UCS del carattere desiderato. La forma può essere usata per i caratteri i cui code point hanno valori decimali pari a `FFFF` o meno, il che include tutti i caratteri del Basic Multilingual Plane.

Per esempio, il code point UCS per la lettera greca  $\beta$  è `000003B2`, e quindi l'universal character name per questo carattere è `\U000003B2` (o `\U000003b2` dato che il caso delle cifre esadecimali non ha importanza). A causa del fatto che le prime quattro cifre esadecimali del code point UCS sono uguali a 0, possiamo anche utilizzare la notazione `\u`, scrivendo il carattere come `\u03B2` o `\u03b2`. I valori code point per l'UCS (che combaciano con quelli dell'Unicode) possono essere trovati all'indirizzo [www.unicode.org/charts/](http://www.unicode.org/charts/).

Non tutti gli universal character name possono essere usati come identificatori. Lo standard C99 contiene una lista di quelli che sono ammessi, un identificatore non può iniziare con un universal character name che rappresenti una cifra.

## 25.5 L'header `<wchar.h>` (C99): utilità per i multibyte estesi e i wide character

L'header `<wchar.h>` fornisce delle funzioni per l'input/output di wide character e per la manipolazione di stringhe di wide character. La vasta maggioranza delle funzioni presenti in `<wchar.h>` sono la versione wide character delle funzioni di altri header (principalmente `<stdio.h>` e `<string.h>`).

L'header `<wchar.h>` dichiara diversi tipi e macro, inclusi i seguenti:

- `mbstate_t` – Un valore di questo tipo può essere usato per salvare lo stato di conversione quando una sequenza di caratteri multibyte viene convertita in una sequenza di wide character o viceversa.
- `wint_t` – Un tipo intero i cui valori rappresentano i caratteri estesi.
- `WEOF` – Una macro che rappresenta un valore `wint_t` che è diverso da qualsiasi altro carattere esteso. `WEOF` viene usato praticamente allo stesso modo di `EOF` [macro `EOF` > 22.2], tipicamente per indicare un errore o la condizione di fine del file.

Osservate che l'header `<wchar.h>` fornisce delle funzioni per i wide character ma non per i caratteri multibyte. Questo perché le normali funzioni di libreria del C sono in grado di gestire i caratteri multibyte e quindi non sono necessarie funzioni speciali. Per esempio, la funzione `fprintf` permette che la sua stringa di formato contenga dei caratteri multibyte.

La maggior parte delle funzioni per i wide character si comporta alla stessa maniera delle funzioni appartenenti alle altre parti della libreria standard. Di solito le uniche modifiche coinvolgono gli argomenti e i valori restituiti che sono del tipo `wchar_t` invece che `char` (oppure `wchar_t *` invece che `char *`). In aggiunta, gli argomenti e i valori restituiti che rappresentano i conteggi per i caratteri sono misurati in wide character invece che in byte. Nel prosieguo di questa sezione indicheremo quale altra funzione (se presente) corrisponde a ogni funzione per i wide character. Non discuteremo ulteriormente delle funzioni per i wide character a meno che non vi siano differenze significative tra questa e la sua controparte "non wide character".

## Orientamento dello stream

Prima di guardare alle funzioni di input/output fornite dall'header `<wchar.h>`, è importante capire un concetto che non esisteva nel C89, ovvero l'**orientamento dello stream**.

Ogni stream è **byte-oriented** (l'orientamento tradizionale) o **wide-oriented** (i dati sono scritti nello stream sotto forma di wide character). Quando uno stream viene aperto per la prima volta, questo non ha nessun orientamento (in particolare, gli stream standard `stdin`, `stdout` e `stderr` non hanno nessun orientamento all'inizio dell'esecuzione del programma [**stream standard > 22.1**]). Eseguire un'operazione sullo stream usando una funzione di input/output fa sì che questo diventi byte-oriented, mentre eseguire un'operazione usando una funzione di input/output per wide character fa sì che diventi wide-oriented. L'orientamento di uno stream può essere selezionato anche chiamando la funzione `fwide` (descritta più avanti in questa sezione). Uno stream mantiene il suo orientamento fino a quando rimane aperto. Chiamare la funzione `freopen` per riaprire lo stream rimuove il suo orientamento [**funzione freopen > 22.2**].

Quando i wide character vengono scritti in uno stream wide-oriented, questi vengono convertiti in caratteri multibyte prima di essere salvati nel file che è associato con lo stream stesso. Viceversa, quando dell'input viene letto da uno stream wide-oriented, i caratteri multibyte trovati nello stream vengono convertiti in wide-character. La codifica multibyte usata in un file è simile a quella usata per i caratteri e per le stringhe all'interno di un programma a eccezione del fatto che le codifiche usate nei file possono contenere dei byte nulli incorporati.

Ogni stream wide-oriented è associato a un oggetto `mbstate_t`, il quale tiene traccia dello stato di conversione dello stream. Quando un wide character scritto in uno stream non corrisponde a nessun carattere multibyte si verifica un errore di codifica. Lo stesso si verifica quando una sequenza di caratteri letti da uno stream non forma un carattere multibyte valido. In entrambi i casi il valore della macro `EILSEQ` (definita nell'header `<errno.h>`) viene salvato nella variabile `errno` per indicare la natura dell'errore [**variabile errno > 24.2**].

Una volta che lo stream è byte-oriented non è possibile applicarvi funzioni di input/output per i wide character. Altre funzioni per gli stream possono essere applicate agli stream di entrambi gli orientamenti, sebbene ci sia qualche particolare considerazione per gli stream wide-oriented.

- Gli stream wide-oriented binari sono soggetti alle restrizioni sul posizionamento nel file sia degli stream testuali che di quelli binari.
- Dopo un'operazione di posizionamento nel file su uno stream wide-oriented una funzione di output di wide character può sovrascrivere parte di un carattere multibyte. Fare questo lascia il resto del file in uno stato indeterminato.
- Chiamare la funzione `fgetpos` [[funzione fgetpos > 22.7](#)] per uno stream wide-oriented recupera l'oggetto `mbstate_t` dello stream come parte dell'oggetto `fpos_t` associato allo stesso stream. Una successiva chiamata alla funzione `fsetpos` [[funzione fsetpos > 22.7](#)] usando il medesimo oggetto `fpos_t` ripristina l'oggetto `mbstate_t` al suo valore precedente.

## Funzioni di input/output formattato per i wide character

```
int fwprintf(FILE * restrict stream,
 const wchar_t * restrict format,
 ...);
int fwscanf(FILE * restrict stream,
 const wchar_t * restrict format,
 ...);
int swprintf(wchar_t * restrict s, size_t n,
 const wchar_t * restrict format,
 ...);
int swscanf(const wchar_t * restrict s,
 const wchar_t * restrict format, ...);
int vfwprintf(FILE * restrict stream,
 const wchar_t * restrict format,
 va_list arg);
int vfwscanf(FILE * restrict stream,
 const wchar_t * restrict format,
 va_list arg);
int vsprintf(wchar_t * restrict s, size_t n,
 const wchar_t * restrict format,
 va_list arg);
int vsscanf(const wchar_t * restrict s,
 const wchar_t * restrict format,
 va_list arg);
int vwprintf(const wchar_t * restrict format,
 va_list arg);
int vwscanf(const wchar_t * restrict format,
 va_list arg);
int wprintf(const wchar_t * restrict format, ...);
```

Le funzioni presenti in questo gruppo sono versioni wide character delle funzioni di input/output che si trovano nell'header <stdio.h> e che sono descritte nella Sezione 22.3. Le funzioni <wchar.h> hanno degli argomenti di tipo `wchar_t *` invece che `char *`, tuttavia il loro comportamento è praticamente lo stesso delle funzioni presenti in <stdio.h>. La Tabella 25.11 illustra la corrispondenza tra le funzioni <stdio.h> e le loro controparti wide character. A meno che non venga detto altrimenti, ogni funzione della colonna sinistra si comporta allo stesso modo della funzione (funzioni) che si trova (trovano) alla sua destra.

Tutte le funzioni di questo gruppo hanno diverse caratteristiche in comune:

- possiedono tutte una stringa di formato la quale consiste di *wide character*,
- le funzioni ...printf, che restituiscono il numero di caratteri scritti, ora restituiscono il conto espresso in *wide character*,
- la specifica di conversione %n si riferisce al numero di *wide character* scritti (nel caso delle funzioni ...printf) o letti (nel caso delle funzioni ...scanf) fino a quel momento.

**Tabella 25.11** Funzioni di input/output formattato per i wide character e le loro equivalenti <stdio.h>

| Funzione <wchar.h>     | Equivalente <stdio.h>            |
|------------------------|----------------------------------|
| <code>fwprintf</code>  | <code>fprintf</code>             |
| <code>fwscanf</code>   | <code>fscanf</code>              |
| <code>swprintf</code>  | <code>sprintf, sprintff</code>   |
| <code>swscanf</code>   | <code>sscanf</code>              |
| <code>vfwprintf</code> | <code>vfprintf</code>            |
| <code>vfwscanf</code>  | <code>vfscanf</code>             |
| <code>vswprintf</code> | <code>vsnprintf, vsprintf</code> |
| <code>vswscanf</code>  | <code>vscanf</code>              |
| <code>vwprintf</code>  | <code>vprintf</code>             |
| <code>vwscanf</code>   | <code>vscanf</code>              |
| <code>wprintf</code>   | <code>printf</code>              |
| <code>wscanf</code>    | <code>scanf</code>               |

`fwprintf` Ulteriori differenze tra la funzione `fwprintf` e la `fprintf` includono:

- lo specificatore di conversione %c viene usato quando l'argomento corrispondente è di tipo `int`. Se è presente il modificatore di lunghezza 1 (rendendo la conversione uguale a %lc), l'argomento viene assunto essere di tipo `wint_t`. In entrambi i casi l'argomento corrispondente viene scritto come un *wide character*;
- lo specificatore di conversione %s viene usato come un puntatore a un *vettore* di caratteri, il quale può contenere dei caratteri multibyte (la `fprintf` non presenta particolari condizioni per i caratteri multibyte). Se è presente il modificatore di lunghezza 1 (come in %ls), l'argomento corrispondente deve essere un *vettore* contenente dei *wide character*. In entrambi i casi, i caratteri presenti nel *vettore* vengono scritti come *wide character* (con la `fprintf` anche la specifica %ls indica un *vettore* di *wide character*, ma questi vengono convertiti in caratteri multibyte prima di essere scritti).

- fwscanf** A differenza della fscanf, la funzione fwscanf legge dei wide character. Le conversioni %c, %s e %[ richiedono una particolare attenzione. Ognuna di queste funzioni legge dei wide character e poi li converte in caratteri multibyte prima di salvarli in un vettore di caratteri. La funzione fwscanf utilizza un oggetto mbstate\_t per tenere traccia dello stato della conversione durante questo processo. L'oggetto viene imposto a zero all'inizio della conversione. Se è presente il modificatore di lunghezza l (facendo diventare le conversioni uguali a %lc, %ls e %l[]), allora i caratteri di input non verranno convertiti, ma piuttosto verranno salvati direttamente in un vettore di elementi wchar\_t. Di conseguenza, quando si legge una stringa di wide character e l'intento è quello di salvarli come wide character, è necessario utilizzare la specifica %ls. Se invece viene usata la specifica %s, i wide character verranno letti dallo stream di input ma convertiti in caratteri multibyte prima di essere salvati.
- swprintf** La funzione swprintf scrive dei wide character all'interno di un vettore con elementi wchar\_t. Questa funzione è simile alla sprintf e alla snprintf, tuttavia non è identica a nessuna delle due. Come la snprintf, anche questa funzione utilizza il parametro n per limitare il numero di caratteri (wide character in questo caso) che andrà a scrivere. Tuttavia, la swprintf restituisce il numero di wide character effettivamente scritti senza includere il carattere null. Sotto questo aspetto sembra la sprintf piuttosto che la snprintf, la quale restituisce il numero di caratteri che avrebbe scritto (carattere null escluso) se non ci fosse stata nessuna restrizione sulla lunghezza. La swprintf restituisce un valore negativo nel caso in cui il numero di wide character che deve essere scritto sia pari a n o più, il che si discosta sia dal comportamento della sprintf che della snprintf.
- vswprintf** La funzione vswprintf è equivalente alla swprintf, con l'argomento arg che sostituisce l'elenco variabile di argomenti della swprintf. Come la swprintf è simile, ma non identica alle funzioni sprintf e snprintf. La funzione vswprintf è una combinazione delle funzioni vsprintf e vsnprintf. Se si cerca di scrivere n o più wide character, la vswprintf restituisce un intero negativo, con un comportamento simile a quello della swprintf.

## Funzioni di input/output per i wide character

```
wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t * restrict s, int n,
 FILE * restrict stream);
wint_t fputwc(wchar_t c, FILE *stream);
int fputws(const wchar_t * restrict s,
 FILE * restrict stream);
int fwipe(FILE *stream, int mode);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c);
wint_t ungetwc(wint_t c, FILE *stream);
```

Le funzioni di questo gruppo sono le versioni wide character delle funzioni di input/output che si trovano in `<stdio.h>` e che sono descritte nella Sezione 22.4. La Tabella 25.12 mostra la corrispondenza tra le funzioni `<stdio.h>` e le loro controparti wide character.

**Tabella 25.12** Funzioni di input/output per i wide character e le loro equivalenti `<stdio.h>`

| Funzione <code>&lt;wchar.h&gt;</code> | Equivalente <code>&lt;stdio.h&gt;</code> |
|---------------------------------------|------------------------------------------|
| <code>fgetwc</code>                   | <code>fgetc</code>                       |
| <code>fgetws</code>                   | <code>fgets</code>                       |
| <code>fputwc</code>                   | <code>fputc</code>                       |
| <code>fputws</code>                   | <code>fputs</code>                       |
| <code>fwide</code>                    | -                                        |
| <code>getwc</code>                    | <code>getc</code>                        |
| <code>getwchar</code>                 | <code>getchar</code>                     |
| <code>putwc</code>                    | <code>putc</code>                        |
| <code>putwchar</code>                 | <code>putchar</code>                     |
| <code>ungetwc</code>                  | <code>ungetc</code>                      |

Come illustrato dalla tabella, la `fwide` è l'unica funzione veramente nuova.

A meno che non venga specificato diversamente, potete assumere che ogni funzione `<wchar.h>` elencata in Tabella 25.12 si comporti come la corrispondente funzione `<stdio.h>`. Tuttavia, c'è una piccola differenza comune a tutte queste funzioni. Per indicare le condizioni di errore o fine del file, alcune funzioni di `<stdio.h>` per l'I/O dei caratteri restituiscono il valore `EOF`. Le equivalenti funzioni `<wchar.h>`, invece, restituiscono il valore `WEOF`.

`fgetwc`  
`getwc`  
`getwchar`  
`fgetws`

C'è un'altra particolarità che riguarda le funzioni di input per i wide character. Una chiamata a una funzione che legge un singolo carattere (`fgetwc`, `getwc` e `getwchar`) può fallire se i byte trovati nello stream di input non formano un wide character valido o se non ci sono abbastanza byte disponibili. Queste condizioni risultano in un errore di codifica, il quale fa sì che la funzione salvi il valore `EILSEQ` nella variabile `errno` e restituisca il valore `WEOF`. La funzione `fgetws`, che legge una stringa di wide character, può incontrare problemi anche a causa di un errore di codifica e in tal caso restituisce un puntatore nullo.

`fputwc`  
`putwc`  
`putwchar`  
`fputws`

Le funzioni per l'output dei wide character possono incontrare anche degli errori di codifica. Nel caso si verificasse un errore di codifica, le funzioni che scrivono un singolo carattere (`fputwc`, `putwc` e `putwchar`), salvano il valore `EILSEQ` in `errno` e restituiscono `WEOF`. Tuttavia, la funzione `fputs` (che scrive una stringa di wide character) è diversa: nel caso si verificasse un errore di codifica restituirebbe il valore `EOF` (e non `WEOF`).

`fwide`

La funzione `fwide` non corrisponde ad alcuna funzione C89. La `fwide` viene usata per determinare l'orientamento corrente di uno stream e, se lo si desidera, cerca di impostarne l'orientamento. Il parametro `mode` determina il comportamento della funzione:

- `mode > 0`. Se lo stream non ha alcun orientamento, cerca di renderlo wide-oriented;

- mode < 0. Se lo stream non ha alcun orientamento, cerca di renderlo byte-oriented;
- mode = 0. L'orientamento non viene modificato.

La funzione `fwide` non modifica l'orientamento dello stream nel caso questo ne avesse già uno.

Il valore restituito dalla `fwide` dipende dall'orientamento dello stream *successivo* alla chiamata. Il valore restituito è positivo se lo stream è wide-oriented, negativo se è byte-oriented e zero se non ha alcun orientamento.

## Utilità generali per le stringhe wide

L'header `<wchar.h>` fornisce un certo numero di funzioni che eseguono delle operazioni sulle stringhe formate da wide character. Queste sono delle versioni wide character delle funzioni che appartengono agli header `<stdlib.h>` e `<string.h>`.

## Funzioni per la conversione numerica delle stringhe wide

```
double wcstod(const wchar_t * restrict nptr,
 wchar_t ** restrict endptr);
float wcstof(const wchar_t * restrict nptr,
 wchar_t ** restrict endptr);
long double wcstold(const wchar_t * restrict nptr,
 wchar_t ** restrict endptr);
long int wcstol(const wchar_t * restrict nptr,
 wchar_t ** restrict endptr,
 int base),
long long int wcstoll(const wchar_t * restrict nptr,
 wchar_t ** restrict endptr,
 int base);
unsigned long int wcstoul(
 const wchar_t * restrict nptr,
 wchar_t ** restrict endptr,
 int base);
unsigned long long int wcstoull(
 const wchar_t * restrict nptr,
 wchar_t ** restrict endptr,
 int base);
```

Le funzioni di questo gruppo sono la versione wide character delle funzioni di conversione numerica presenti in `<stdlib.h>` e descritte nella Sezione 26.2. Le funzioni `<wchar.h>` possiedono degli argomenti di tipo `wchar_t *` e `wchar_t **` invece che `char *` e `char **`, ma il loro comportamento è essenzialmente lo stesso di quello delle funzioni `<stdlib.h>`. La Tabella 25.13 illustra la corrispondenza tra le funzioni `<stdlib.h>` e le loro controparti wide character.

**Tabella 25.13** Funzioni di conversione numerica per le stringhe wide e le loro equivalenti <stdlib.h>

| Funzione <wchar.h> | Equivalenti <stdlib.h> |
|--------------------|------------------------|
| wcstod             | strtod                 |
| wcstof             | strtof                 |
| wcstold            | strtold                |
| wcstol             | strtol                 |
| wcstoll            | strtoll                |
| wcstoul            | strtoul                |
| wcstoull           | strtoull               |

## Funzioni per la copia di stringhe wide

```
wchar_t *wcscpy(wchar_t * restrict s1,
 const wchar_t * restrict s2);
wchar_t *wcsncpy(wchar_t * restrict s1,
 const wchar_t * restrict s2,
 size_t n);
wchar_t *wmemcpy(wchar_t * restrict s1,
 const wchar_t * restrict s2,
 size_t n),
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2,
 size_t n);
```

Le funzioni di questo gruppo sono le versioni wide character delle funzioni per la copia delle stringhe che si trovano nell'header <string.h> e che sono descritte nella Sezione 23.6. Le funzioni <wchar.h> hanno degli argomenti wchar\_t \* invece che char \*, ma il loro comportamento è essenzialmente lo stesso delle funzioni <string.h>. La Tabella 25.14 illustra la corrispondenza tra le funzioni <string.h> e le loro controparti wide character.

**Tabella 25.14** Funzioni per la copia delle stringhe wide e le loro equivalenti <string.h>

| Funzione <wchar.h> | Equivalenti <string.h> |
|--------------------|------------------------|
| wcscpy             | strcpy                 |
| wcsncpy            | strncpy                |
| wmemcpy            | memcpy                 |
| wmemmove           | memmove                |

## Funzioni per la concatenazione delle stringhe wide

```
wchar_t *wcscat(wchar_t * restrict s1,
 const wchar_t * restrict s2);
wchar_t *wcsncat(wchar_t * restrict s1,
 const wchar_t * restrict s2,
 size_t n);
```

Le funzioni di questo gruppo sono le versioni wide character delle funzioni per la concatenazione delle stringhe che si trovano nell'header <string.h> e che sono descritte nella Sezione 23.6. Le funzioni <wchar.h> hanno degli argomenti `wchar_t *` invece che `char *`, ma il loro comportamento è essenzialmente lo stesso delle funzioni <string.h>. La Tabella 25.15 illustra la corrispondenza tra le funzioni <string.h> e le loro controparti wide character.

**Tabella 25.15** Funzioni per la concatenazione delle stringhe wide e le loro equivalenti <string.h>

| Funzione <wchar.h>    | Equivalenti <string.h> |
|-----------------------|------------------------|
| <code>wcscpy</code>   | <code>strncpy</code>   |
| <code>wcsncpy</code>  | <code>strncpy</code>   |
| <code>wmemcpy</code>  | <code>memncpy</code>   |
| <code>wmemmove</code> | <code>memmove</code>   |

### Funzioni per il confronto delle stringhe wide

```
int wcscmp(const wchar_t *s1, const wchar_t *s2),
int wcsncmp(const wchar_t *s1, const wchar_t *s2,
 size_t n),
size_t wcsxfrm(wchar_t * restrict s1,
 const wchar_t * restrict s2,
 size_t n),
int wmemcmp(const wchar_t *s1, const wchar_t *s2,
 size_t n);
```

Le funzioni di questo gruppo sono le versioni wide character delle funzioni per il confronto delle stringhe che si trovano nell'header <string.h> e che sono descritte nella Sezione 23.6. Le funzioni <wchar.h> hanno degli argomenti `wchar_t *` invece che `char *`, ma il loro comportamento è essenzialmente lo stesso delle funzioni <string.h>. La Tabella 25.16 illustra la corrispondenza tra le funzioni <string.h> e le loro controparti wide character.

**Tabella 25.16** Funzioni per il confronto delle stringhe wide e le loro equivalenti <string.h>

| Funzione <wchar.h>   | Equivalenti <string.h> |
|----------------------|------------------------|
| <code>wcscmp</code>  | <code>strcmp</code>    |
| <code>wcsncmp</code> | <code>strcoll</code>   |
| <code>wcsncmp</code> | <code>strncmp</code>   |
| <code>wcsxfrm</code> | <code>strxfrm</code>   |
| <code>wmemcmp</code> | <code>memcmp</code>    |

## Funzioni per la ricerca nelle stringhe wide

```
wchar_t *wcschr(const wchar_t *s, wchar_t c);
size_t wcsccspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcspbrk(const wchar_t *s1,
 const wchar_t *s2);
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
size_t wcspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsstr(const wchar_t *s1,
 const wchar_t *s2);
wchar_t *wcstok(wchar_t * restrict s1,
 const wchar_t * restrict s2,
 wchar_t ** restrict ptr);
wchar_t *wmemchr(const wchar_t *s, wchar_t c,
 size_t n);
```

Le funzioni di questo gruppo sono le versioni wide character delle funzioni per la ricerca nelle stringhe che si trovano nell'header <string.h> e che sono descritte nella Sezione 23.6. Le funzioni <wchar.h> hanno degli argomenti `wchar_t *` e `wchar_t **` invece che `char *` e `char **`, ma il loro comportamento è essenzialmente lo stesso delle funzioni <string.h>. La Tabella 25.17 illustra la corrispondenza tra le funzioni <string.h> e le loro controparti wide character.

**Tabella 25.17** Funzioni per la ricerca nelle stringhe wide e le loro equivalenti <string.h>

| Funzione <wchar.h>    | Equivalente <string.h> |
|-----------------------|------------------------|
| <code>wcschr</code>   | <code>strchr</code>    |
| <code>wcsccspn</code> | <code>strccspn</code>  |
| <code>wcspbrk</code>  | <code>strbrk</code>    |
| <code>wcsrchr</code>  | <code>strrchr</code>   |
| <code>wcspn</code>    | <code>strspn</code>    |
| <code>wcsstr</code>   | <code>strstr</code>    |
| <code>wcstok</code>   | <code>strtok</code>    |
| <code>wmemchr</code>  | <code>memchr</code>    |

**wcstok** La funzione `wcstok` serve allo stesso scopo della funzione `strtok` ma viene usata in modo diverso grazie al suo terzo parametro (la `strtok` ha solamente due parametri). Per capire come funziona la `wcstok`, abbiamo prima bisogno di rivedere il comportamento della `strtok`.

Nella Sezione 23.6 abbiamo visto che la funzione `strtok` va alla ricerca di un “token” (una sequenza di caratteri che non includono certi caratteri di delimitazione) all'interno di una stringa: La chiamata `strtok(s1, s2)` scansiona la stringa `s1` alla ricerca di una sequenza non vuota di caratteri che *non* sono presenti nella stringa `s2`. La `strtok` segnala la fine di un token mettendo un carattere null nella stringa `s1` immediatamente dopo l'ultimo carattere del token stesso. La funzione restituisce un puntatore al primo carattere presente nel token.

Le successive chiamate alla strtok possono trovare dei token aggiuntivi all'interno della stessa stringa. La chiamata strtok(NULL, s2) continua la ricerca iniziata dalla chiamata precedente. Come prima, la strtok segna la fine di un token con il carattere null e poi restituisce un puntatore all'inizio del token stesso. Il processo può essere ripetuto fino a quando la strtok restituisce un puntatore nullo, indicando che non è stato trovato nessun token.

Uno dei problemi con la funzione strtok è che questa utilizza una variabile statica per tenere traccia della ricerca, il che rende impossibile usare questa funzione per effettuare delle ricerche simultanee su due o più stringhe. Grazie al suo parametro aggiuntivo, la wcstok non presenta questo problema.

I primi due parametri della wcstok sono gli stessi della strtok (a eccezione del fatto che questi puntano a stringhe di wide character). Il terzo argomento, prt, punta a una variabile di tipo wchar\_t \*. La funzione salverà in questa variabile le informazioni che permetteranno alle chiamate successive alla wcstok di continuare la scansione della stessa stringa (quando il primo argomento è un puntatore nullo). Quando la ricerca viene ripresa da una successiva invocazione alla wcstok, un puntatore alla stessa variabile dovrà essere passato come terzo argomento. Il valore di questa variabile non deve essere modificato tra due chiamate alla wcstok.

Per vedere come lavora questa funzione, rifacciamo l'esempio della Sezione 23.6. Assumete che str, p e q siano dichiarate in questo modo:

```
wchar_t str[] = L" April 28,1998";
wchar_t *p, *q;
```

La nostra chiamata iniziale alla funzione wcstok passerà str come primo argomento:

```
p = wcstok(str, L"\t", &q);
```

ora p punta al primo carattere della parola April, il quale è seguito da un wide character null. Chiamare la wcstok con un puntatore nullo come primo argomento e &q come terzo argomento farà riprendere la ricerca da dove era stata interrotta:

```
p = wcstok(NULL, L"\t", &q);
```

Dopo questa chiamata p punta al primo carattere di 28, che adesso è terminato da un carattere wide null. Un'ultima chiamata alla funzione wcstok individua l'anno:

```
p = wcstok(NULL, L"\t", &q);
```

ora p punta al primo carattere presente di 1998.

## Funzioni varie

```
-size_t wcslen(const wchar_t *s),
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n),
```

Le funzioni di questo gruppo sono le versioni wide character delle varie funzioni per le stringhe che si trovano nell'header <string.h> e che sono descritte nella Sezione 23.6. Le funzioni <wchar.h> hanno degli argomenti wchar\_t \* invece che char \*, ma il loro comportamento è essenzialmente lo stesso delle funzioni <string.h>. La Tabella 25.18 illustra la corrispondenza tra le funzioni <string.h> e le loro controparti wide character.

**Tabella 25.18** Funzioni varie per le stringhe wide e le loro equivalenti <string.h>

| Funzione <wchar.h> | Equivalenti <string.h> |
|--------------------|------------------------|
| wcslen             | strlen                 |
| wmemset            | memset                 |

## Funzioni wide character per la conversione degli orari

```
size_t wcsftime(wchar_t * restrict s, size_t maxsize,
 const wchar_t * restrict format,
 const struct tm * restrict tmpt);
```

wcsftime

La funzione wcsftime è la versione wide character della strftime, la quale appartiene all'header <time.h> ed è descritta nella Sezione 26.3.

## Utilità per la conversione multibyte esteso/wide character

Ora esaminiamo le funzioni di <wchar.h> che eseguono le conversioni tra i caratteri multibyte e i wide character. Cinque di queste (nbrlen, mbrtowc, wcrtomb, mbsrtowcs e wcsrtombs) corrispondono alle funzioni di conversione dal formato multibyte a quello wide character (e viceversa) per i caratteri e le stringhe dichiarate che sono in <stdlib.h>. Le funzioni <wchar.h> possiedono un parametro aggiuntivo, ovvero un puntatore a una variabile di tipo mbstate\_t. Questa variabile tiene traccia dello stato della conversione di una sequenza di caratteri multibyte in una sequenza di wide character (o viceversa) basata sulla localizzazione corrente. Ne risulta che le funzioni <wchar.h> sono "riavvibili", passando a queste un puntatore a una variabile mbstate\_t modificata da una precedente chiamata a funzione, possiamo "riavviarla" usando lo stato della conversione da quella chiamata. Un vantaggio di questo arrangemento è che permette a due funzioni di condividere lo stesso stato di conversione. Per esempio, delle chiamate alla mbrtowc e alla mbsrtowcs usate per elaborare una singola stringa di caratteri multibyte possono condividere una variabile mbstate\_t.

Lo stato di conversione contenuto in una variabile mbstate\_t è caratterizzato dallo stato di shift corrente e dalla posizione corrente in un carattere multibyte. Impostare a zero i byte di una variabile mbstate\_t la pone nello stato di conversione iniziale, il che significa che nessun carattere multibyte è già in elaborazione e che lo stato di shift iniziale è attivo:

```
mbstate_t state;
...
memset(&state, '\0', sizeof(state));
```

Passare &state a una delle funzioni riavvibili fa sì che la conversione inizi con lo stato di conversione iniziale. Una volta che la variabile mbstate\_t è stata alterata da una di queste funzioni, non deve essere usata per convertire una sequenza di caratteri multibyte diversa e non deve nemmeno essere usata per eseguire una conversione

nella direzione opposta. Cercare di eseguire una di queste azioni risulterebbe in comportamento indefinito. Anche usare la variabile dopo una modifica alla categ LC\_CTYPE di una localizzazione provoca un comportamento indefinito.

## Funzioni di conversione tra byte singoli e wide character

```
wint_t btowc(int c);
int wctob(wint_t c);
```

Le funzioni di questo gruppo convertono dei caratteri rappresentati con un singolo byte in wide character e viceversa.

- btowc** La funzione btowc restituisce WEOF se c è uguale a EOF oppure se c (quando viene aperto un cast ad unsigned char) non è un carattere a singolo byte valido nello stato di shift iniziale. Negli altri casi la funzione restituisce la rappresentazione wide character del suo argomento.
- wctob** La funzione wctob è l'opposto della btowc. Restituisce EOF se c non corrisponde a un carattere multibyte nello stato di shift iniziale. Negli altri casi, questa funzione restituisce la rappresentazione di c su singolo byte.

## Funzioni di conversione dello stato

```
int mbsinit(const mbstate_t *ps);
```

- mbsinit** Questo gruppo consiste di una singola funzione, che restituisce un valore diverso da zero nel caso in cui ps sia un puntatore nullo o punti a una variabile mbstate\_t contenente la descrizione dello stato di conversione iniziale.

## Funzioni riavviable per la conversione tra caratteri multibyte e wide character

```
size_t mbrlen(const char * restrict s, size_t n,
 mbstate_t * restrict ps);
size_t mbrtowc(wchar_t * restrict pwc,
 const char * restrict s, size_t n,
 mbstate_t * restrict ps);
size_t wcrtomb(char * restrict s, wchar_t wc,
 mbstate_t * restrict ps);
```

Le funzioni di questo gruppo sono le versioni riavviable delle funzioni mblen, mbrcpy e wcrtomb, le quali appartengono all'header <stdlib.h> e sono discusse nella Sezione 25.2. Le nuove funzioni mbrlen, mbrtowc e wcrtomb differiscono dalle loro controparti <stdlib.h> sotto diversi aspetti.

- Le funzioni mbrlen, mbrtowc e wcrtomb possiedono un parametro aggiuntivo chiamato ps. Quando una di queste viene chiamata, l'argomento corrispondente deve puntare a una variabile di tipo mbstate\_t. La funzione salverà lo stato della corrispondente versione all'interno di questa variabile. Se l'argomento corrispondente a ps è

puntatore nullo, la funzione userà una variabile interna per salvare lo stato della conversione (all'inizio dell'esecuzione del programma questa variabile viene impostata allo stato di conversione iniziale).

- Quando il parametro *s* è un puntatore nullo, le vecchie funzioni *mblen*, *mbtowc* e *wctomb* restituiscono un valore diverso da zero se le codifiche dei caratteri multibyte sono dipendenti dallo stato (altrimenti restituiscono il valore zero). Le nuove funzioni non seguono questo comportamento.
- Le funzioni *mbrlen*, *mbrtowc* e *wcrtomb* restituiscono un valore di tipo *size\_t* invece che *int* (il tipo restituito dalle vecchie funzioni).

**mbrlen** Una chiamata alla funzione *mbrlen* è equivalente alla chiamata

*mbrtowc(NULL, s, n, ps)*

a eccezione del caso in cui *ps* sia un puntatore nullo. In tal caso viene usato l'indirizzo di una variabile interna.

**mbrtowc** Se *s* è un puntatore nullo, una chiamata alla *mbrtowc* è equivalente alla chiamata  
*mbrtowc(NULL, "", 1, ps)*

Negli altri casi, una chiamata alla funzione *mbrtowc* esamina fino a *n* byte puntati da *s* per vedere se questi completano un carattere multibyte valido (notate che un carattere multibyte può essere già in elaborazione prima della chiamata, così come tracciato dalla variabile *mbstate\_t* puntata da *ps*). In tal caso quei byte vengono convertiti in un wide character. Il wide character viene salvato nella locazione puntata da *pwc* ammesso che questo non sia uguale a null. Se questo carattere è il wide character null, la variabile *mbstate\_t* usata durante la chiamata viene lasciata allo stato di conversione iniziale.

La funzione *mbrtowc* possiede diversi possibili valori restituiti. Restituisce uno 0 se la conversione produce un wide character null. Restituisce un numero compreso tra 1 ed *n* se la conversione produce un wide character diverso da quello null. In tal caso il valore restituito rappresenta il numero di byte usati per completare il carattere multibyte. Restituisce -2 se gli *n* byte puntati da *s* non sono sufficienti per completare un carattere multibyte (sebbene i byte stessi fossero validi). Infine, restituisce -1 se si verifica un errore di codifica (la funzione incontra i byte che non formano un carattere multibyte valido). Nell'ultimo caso, la funzione si occupa anche di salvare il valore EILSEQ nella variabile *errno*.

**wcrtomb** Se *s* è un puntatore nullo, una chiamata alla *wcrtomb* è equivalente alla chiamata  
*wcrtomb(buf, L'\0', ps)*

dove *buf* è un buffer interno. Negli altri casi la *wcrtomb* converte *wc* da wide character a carattere multibyte, il quale viene salvato nel vettore puntato da *s*. Se *wc* corrisponde al wide character null, la funzione salva un byte null preceduto da una sequenza di shift, se questa è necessaria a ripristinare lo stato di shift iniziale. In questo caso la variabile *mbstate\_t* usata durante la chiamata viene lasciata allo stato di conversione iniziale. La *wcrtomb* restituisce il numero di byte che salva, includendo le sequenze di shift. Se *wc* non corrisponde a un wide character valido, la funzione restituisce -1 e salva il valore EILSEQ in *errno*.

## Funzioni riavviable di conversione tra stringhe multibyte e stringhe wide

```
size_t mbsrtowcs(wchar_t * restrict dst,
 const char ** restrict src,
 size_t len,
 mbstate_t * restrict ps);
size_t wcsrtombs(char * restrict dst,
 const wchar_t ** restrict src,
 size_t len,
 mbstate_t * restrict ps);
```

`mbsrtowcs`  
`wcsrtombs`

Le funzioni `mbsrtowcs` e `wcsrtombs` sono le versioni riavviable delle funzioni `mbstowcs` e `wcstombs` che appartengono all'header `<stdlib.h>` e sono discusse nella Sezione 25.2. La `mbsrtowcs` e la `wcsrtombs` sono essenzialmente uguali alle loro controparti `<stdlib.h>` ma differiscono da queste per le seguenti ragioni.

- Le funzioni `mbsrtowcs` e `wcsrtombs` possiedono un parametro aggiuntivo chiamato `ps`. Quando una di queste viene chiamata, l'argomento corrispondente deve puntare a una variabile di tipo `mbstate_t`. La funzione salverà lo stato della conversione in questa variabile (all'inizio dell'esecuzione del programma, questa variabile viene impostata allo stato di conversione iniziale). Entrambe le funzioni aggiornano lo stato mentre la conversione è in atto. Se la conversione si interrompe a causa del raggiungimento di un carattere null, la variabile `mbstate_t` viene lasciata allo stato di conversione iniziale.
- Per le funzioni `mbsrtowcs` e `wcsrtombs`, il parametro `src`, che rappresenta un vettore contenente i caratteri che devono essere convertiti (il vettore sorgente) è un puntatore a un puntatore (nelle vecchie funzioni `mbstowcs` e `wcstombs` il parametro corrispondente era semplicemente un puntatore). Questa modifica permette a queste funzioni di tenere traccia del punto nel quale la conversione si è interrotta. Il puntatore puntato da `src` viene impostato al valore null nel caso in cui la conversione si interrompa a causa del raggiungimento di un carattere null. Negli altri casi il puntatore viene impostato in modo che punti immediatamente dopo l'ultimo carattere convertito.
- Il parametro `dst` può essere un puntatore nullo, in tal caso i caratteri convertiti non vengono salvati e il puntatore puntato da `src` non viene modificato.
- Quando una di queste funzioni incontra un carattere non valido nel vettore sorgente, salva il valore `EILSEQ` nella variabile `errno` (in aggiunta restituisce il valore `-1` esattamente come le vecchie funzioni).

## 25.6 L'header <wctype.h> (C99): utilità per la classificazione e la mappatura dei wide character

L'header <wctype.h> è la versione wide character dell'header <ctype.h> [header <ctype.h> 23.5]. L'header <ctype.h> fornisce due tipi di funzioni: quelle per la classificazione dei caratteri (come la `isdigit`, che controlla se un carattere corrisponde a una cifra) e quelle di case-mapping (come la `toupper` che converte una lettera minuscola in una lettera maiuscola). L'header <wctype.h> fornisce funzioni simili per i wide character, sebbene differisca da <ctype.h> per un fatto importante: alcune delle funzioni presenti in <wctype.h> sono "estensibili", ovvero possono effettuare delle classificazioni o delle mappature definite dal programmatore.

L'header <wctype.h> dichiara tre tipi e una macro. Il tipo `wint_t` e la macro `WEOF` sono discusse nella Sezione 25.5. I tipi rimanenti sono `wctype_t`, i cui valori rappresentano delle classificazioni dei caratteri specifiche per una localizzazione, e `wctrans_t`, i cui valori rappresentano delle mappature per i caratteri specifiche per una data localizzazione.

La maggior parte delle funzioni presenti in <wctype.h> richiedono un argomento `wint_t`. Il valore di questo argomento deve essere un wide character (un valore `wchar_t`) oppure `WEOF`. Passare un qualsiasi altro argomento è causa di un comportamento indefinito.

Il comportamento delle funzioni presenti in <wctype.h> è dipendente dalla categoria `LC_CTYPE` della localizzazione corrente.

### Funzioni di classificazione dei wide character

```
int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswblank(wint_t wc);
int iswcntrl(wint_t wc);
int iswdigit(wint_t wc);
int iswgraph(wint_t wc);
int iswlower(wint_t wc);
int iswprint(wint_t wc);
int iswpunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswdxdigit(wint_t wc);
```

Ogni funzione per la classificazione dei wide character restituisce un valore diverso da zero se il suo argomento rispetta una particolare caratteristica. La Tabella 25.19 elenca le proprietà di ogni insieme di funzioni.

Le descrizioni della Tabella 25.19 trascurano alcune sottigliezze sui wide character. Per esempio, la definizione della funzione `iswgraph` nello standard C99 asserisce che questa "verifica se il wide character è tale per cui la funzione `iswprint` è vera e la

`iswspace` è falsa", lasciando così aperta la possibilità che più di un wide character considerato come uno "spazio". Per una descrizione dettagliata di queste leggete l'Appendice D.

Nella maggior parte dei casi le funzioni per la classificazione dei wide character sono coerenti con le corrispondenti funzioni di `<ctype.h>`: se una funzione `h` restituisce un valore diverso da zero (indicando il valore "vero"), allora la corrispondente funzione `<wctype.h>` restituisce un valore vero per la versione wide dello stesso carattere. L'unica eccezione coinvolge i wide character di spazio bianco (dallo spazio) che sono anche dei caratteri stampabili, i quali possono essere classificati in modo diverso dalle funzioni `iswgraph` e `iswpunct` rispetto alle funzioni `isgraph` e `ispunct`. Per esempio, un carattere per il quale `isgraph` restituisce un valore vero non farà restituire un valore falso dalla funzione `iswgraph`.

**Tabella 25.19** Funzioni per la classificazione dei wide character

| Nome                        | Controllo                                             |
|-----------------------------|-------------------------------------------------------|
| <code>iswalnum(wc)</code>   | wc è un carattere alfanumerico?                       |
| <code>iswalpha(wc)</code>   | wc è una lettera?                                     |
| <code>iswblank(wc)</code>   | wc è un carattere vuoto? <sup>†</sup>                 |
| <code>iswcntrl(wc)</code>   | wc è un carattere di controllo?                       |
| <code>iswdigit(wc)</code>   | wc è una cifra decimale?                              |
| <code>iswgraph(wc)</code>   | wc è un carattere stampabile (diverso da uno spazio)? |
| <code>iswlower(wc)</code>   | wc è una lettera minuscola?                           |
| <code>iswprint(wc)</code>   | wc è un carattere stampabile (incluso lo spazio)?     |
| <code>iswpunct(wc)</code>   | wc è un carattere di punteggiatura?                   |
| <code>iswspace(wc)</code>   | wc è un carattere di spazio bianco?                   |
| <code>iswupper(wc)</code>   | wc è una lettera maiuscola?                           |
| <code>iswdxdigit(wc)</code> | wc è una cifra esadecimale?                           |

<sup>†</sup>I wide character vuoti standard sono lo spazio (`L' '` ) e la tabulazione orizzontale (`L'\t`)

## Funzioni estendibili di classificazione dei wide character

```
int iswctype(wint_t wc, wctype_t desc),
wctype_t wctype(const char *property);
```

Ogni funzione di classificazione dei wide character appena discussa è in grado di verificare una singola condizione prefissata. Le funzioni `wctype` e `iswctype`, che sono state progettate per essere usate assieme, rendono possibile controllare altre condizioni con la stessa modalità.

`wctype` Se alla funzione `wctype` viene passata una stringa che descrive una classe di wide character, questa restituisce un valore che rappresenta quest'ultima. Per esempio, la chiamata

```
wctype("upper")
```

raacter sia  
funzioni  
character  
e <ctype.  
la corri-  
ide dello  
(diversi  
classificati  
sgraph e  
vero può

**iswctype**

restituisce un valore `wctype_t` rappresentante la classe delle lettere maiuscole. Lo standard C99 richiede che per la funzione `wctype` siano ammesse come argomento i seguenti stringhe:

```
"alnum" "alpha" "blank" "cntrl" "digit" "graph"
"lower" "print" "punct" "space" "upper" "xdigit"
```

Delle stringhe aggiuntive possono essere fornite dall'implementazione. Quali stringhe siano ammissibili per la funzione `wctype` in un dato momento dipende dalla categoria `LC_CTYPE` della localizzazione corrente. Le 12 stringhe sopra elencate sono ammesse in tutte le localizzazioni. Se a `wctype` viene passata una stringa che non è supportata dalla localizzazione corrente, questa restituisce uno zero.

Una chiamata alla funzione `iswctype` richiede due parametri: `wc` (un wide character) `desc` (un valore restituito da `wctype`). La funzione `iswctype` restituisce un valore diverso da zero nel caso in cui `wc` appartenga alla classe di caratteri corrispondente a `desc`. Per esempio, la chiamata

```
iswctype(wc, wctype("alnum"))
```

è equivalente alla chiamata

```
iswalnum(wc)
```

Le funzioni `wctype` e `iswctype` sono le più utili quando l'argomento di `wctype` è una stringa diversa da quelle standard sopra elencate.

## Funzioni di mappatura dei wide character

**towlower**  
**toupper**

```
wint_t towlower(wint_t wc);
wint_t toupper(wint_t wc);
```

Le funzioni `towlower` e `toupper` sono le controparti wide character delle funzioni `tolower` e `toupper`. Per esempio, la `towlower` restituisce la versione minuscola del suo argomento nel caso questo sia una lettera maiuscola, altrimenti lo restituisce senza modificarlo. Come al solito possono esserci delle stranezze quando si ha a che fare con i wide character. Per esempio, nella corrente localizzazione può esistere più di una versione minuscola di una lettera, in tal caso la funzione `towlower` può restituire una qualsiasi di queste.

## Funzioni estendibili per la mappatura dei wide character

```
wint_t towctrans(wint_t wc, wctrans_t desc);
wctrans_t wctrans(const char *property);
```

Le funzioni `wctrans` e `towctrans` vengono utilizzate insieme per supportare una mappatura generalizzata dei wide character.

wctrans Alla funzione wctrans viene passata una stringa che descrive una mappatura dei caratteri e questa restituisce un valore wctrans\_t che rappresenta la mappatura stessa. Per esempio, la chiamata

```
wctrans("tolower")
```

restituisce un valore wctrans\_t rappresentante la mappatura delle lettere maiuscole in quelle minuscole. Lo standard C99 richiede che le stringhe "tolower" e "toupper" siano ammesse come argomenti alla wctrans. Delle stringhe aggiuntive possono essere fornite dall'implementazione. Quali stringhe siano ammissibili come argomenti in un dato momento dipende dalla categoria LC\_CTYPE della localizzazione corrente. Le stringhe "tolower" e "toupper" sono ammissibili in tutte le localizzazioni. Se alla funzione wctrans viene passata una stringa che non è supportata dalla corrente localizzazione, questa restituisce il valore zero.

Una chiamata alla funzione towctrans richiede due parametri: wc (un wide character) e desc (un valore restituito dalla funzione wctrans). La towctrans mappa wc in un altro wide character basandosi sulla mappatura specificata da desc. Per esempio, la chiamata

```
towctrans(wc, wctrans("tolower"))
```

è equivalente alla

```
tolower(wc)
```

La funzione towctrans è utile principalmente se usata congiuntamente alle mappature definite dall'implementazione.

## Domande & Risposte

**D: Quanto lunga è la stringa con le informazioni sulla localizzazione restituita dalla funzione setlocale? [p. 666]**

R: Non c'è una lunghezza massima e questo solleva una domanda: come possiamo riservare dello spazio per la stringa se non sappiamo quanto questa sia lunga? La risposta naturalmente è l'allocazione dinamica della memoria. Il seguente frammento di programma (basato su un esempio simile presente nel libro di Harbinson e Steel C: *A Reference Manual*) mostra come determinare il quantitativo di memoria necessario, allocare dinamicamente la memoria e poi copiare le informazioni sulla localizzazione in quella memoria:

```
char *temp, *old_locale;
temp = setlocale(LC_ALL, NULL);
if (temp == NULL) {
 /* informazioni sulla localizzazione non disponibili */
}
old_locale = malloc(strlen(temp) + 1);
if (old_locale == NULL) {
 /* allocazione della memoria non andata a buon fine */
}
strcpy(old_locale, temp);
```

Ora possiamo passare a una localizzazione diversa e poi ripristinare in un secondo momento la vecchia localizzazione.

```
setlocale(LC_ALL, ""); /* passa alla localizzazione nativa */

setlocale(LC_ALL, old_locale); /* ripristina la vecchia localizzazione */
```

**D: Perché il C prevede sia i caratteri multibyte che wide character? Uno dei due tipi non sarebbe stato sufficiente? [p. 670]**

**R:** Le due codifiche servono a scopi diversi. I caratteri multibyte sono comodi per l'input/output visto che i dispositivi di I/O spesso sono byte-oriented. I wide character, d'altro canto, sono più convenienti per lavorare all'interno di un programma visto che ogni wide character occupa la stessa quantità di spazio. Quindi un programma può leggere dei caratteri multibyte, convertirli in wide character per poterli manipolare al suo interno e poi convertirli nuovamente nel formato multibyte per effettuare l'output.

**D: L'Unicode e l'UCS sembrano praticamente uguali. Qual è la differenza tra i due? [p. 672]**

**R:** Entrambi contengono gli stessi caratteri e i caratteri sono rappresentati dagli stessi code point. L'Unicode, però, è più di un semplice set di caratteri. Per esempio, l'Unicode supporta "l'ordine di visualizzazione bidirezionale". Alcuni linguaggi incluso l'arabo e l'ebraico permettono al testo di essere scritto da destra a sinistra invece che da sinistra a destra. L'Unicode è in grado di specificare l'ordine di visualizzazione dei caratteri permettendo al testo di contenere alcuni caratteri che devono essere visualizzati da sinistra a destra assieme ad altri che vanno da destra a sinistra.

## Esercizi

### Sezione 25.1

1. Determinate quali localizzazioni sono supportate dal vostro compilatore.

### Sezione 25.2

2. La codifica Shift-JIS per il *kanji* richiede uno o due byte per carattere. Se il primo byte di un carattere è compreso tra 0x81 e 0x9f o tra 0xe0 e 0xef, viene richiesto un altro byte (qualsiasi altro byte viene trattato come un carattere a se stante). Il secondo byte deve essere compreso tra 0x40 e 0x7e o tra 0x80 e 0xfc (tutti gli intervalli sono da considerarsi chiusi). Per ognuna delle seguenti stringhe, date il valore che la funzione `mbcheck` della Sezione 25.2 restituirebbe se la stringa venisse passata come argomento. Assumete che i caratteri multibyte siano codificati usando la codifica Shift-JIS nella localizzazione corrente.
  - (a) "\x05\x87\x80\x36\xed\xaa"
  - (b) "\x20\xe4\x50\x88\x3f"
  - (c) "\xde\xad\xbe\xef"
  - (d) "\x8a\x60\x92\x74\x41"
3. Una delle proprietà utili del UTF-8 è che nessuna sequenza di caratteri multibyte può rappresentare un altro carattere multibyte valido. La codifica Shift-JIS per il *kanji* (discussa nell'Esercizio 2) possiede la stessa proprietà?

4. Fornite una stringa letterale C che rappresenti ognuna delle seguenti frasi. Assumete che i caratteri à, è, ê, ï, ô, û e ü siano rappresentati da caratteri a singolo byte della codifica Latin-1 (avrete bisogno di cercare i code point Latin-1 per questi caratteri). Per esempio, la frase *déjà vu* può essere rappresentata con la stringa "d\xe9j\xeu vu".
  - (a) *Côte d'Azur*
  - (b) *crème brûlée*
  - (c) *crème fraîche*
  - (d) *Fahvergnügen*
  - (e) *tête-à-tête*
5. Ripetete l'Esercizio 4, questa volta usando la codifica multibyte UTF-8. Per esempio, la frase *déjà vu* può essere rappresentata dalla stringa "d\xc3\xa9j\xc3\xa0 vu".
6. Modificate il seguente frammento di programma sostituendo quanti più caratteri possibile con dei trigrafi.

```
while ((orig_char = getchar()) != EOF) {
 new_char = orig_char ^ KEY;
 if (isprint(orig_char) && isprint(new_char))
 putchar(new_char);
 else
 putchar(orig_char);
}
```

7. Modificate il frammento di programma dell'Esercizio 6 sostituendo quanti più token possibile con i digrafi e le macro definite in `<iso646.h>`.

## Progetti di programmazione

1. Scrivete un programma che controlli se la localizzazione "" (nativa) del vostro compilatore sia uguale alla localizzazione "C".
2. Scrivete un programma che ottenga il nome di una localizzazione dalla riga di comando e poi visualizzi i valori contenuti dalla corrispondente struttura `lconv`. Per esempio, se la localizzazione è "fi\_FI" (Finlandia), l'output del programma può somigliare al seguente:

```
decimal_point = ","
thousands_sep = " "
grouping = 3
mon_decimal_point = ","
mon_thousands_sep = " "
mon_grouping = 3
positive_sign = ""
negative_sign = "-"
```

```
currency_symbol = "EUR"
frac_digits = 2
p_cs_precedes = 0
n_cs_precedes = 0
p_sep_by_space = 2
n_sep_by_space = 2
p_sign_posn = 1
n_sign_posn = 1
int_curr_symbol = "EUR "
int_frac_digits = 2
int_p_cs_precedes = 0
int_n_cs_precedes = 0
int_p_sep_by_space = 2
int_n_sep_by_space = 2
int_p_sign_posn = 1
int_n_sign_posn = 1
```

Per questioni di leggibilità, i caratteri di grouping e mon\_grouping devono essere visualizzati come numeri decimali.



# 26 Funzioni di libreria

Gli unici header che non sono stati trattati nei precedenti capitoli, `<stdarg.h>`, `<stdlib.h>` e `<time.h>`, sono diversi da tutti gli altri presenti nella libreria standard. L'header `<stdarg.h>` (Sezione 26.1) rende possibile la scrittura di funzioni con un numero variabile di argomenti. L'header `<stdlib.h>` (Sezione 26.2) è un assortimento di funzioni che non rientrano in nessuno degli altri header. L'header `<time.h>` (Sezione 26.3) permette ai programmi di lavorare con le date e le ore.

## 26.1 L'header `<stdarg.h>`: argomenti variabili

```
type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, parmN);
```

Le funzioni come la `printf` e la `scanf` possiedono una proprietà insolita: ammettono un numero qualsiasi di argomenti. L'abilità di gestire un numero variabile di argomenti non è limitata solamente alle funzioni di libreria. L'header `<stdarg.h>` fornisce strumenti di cui abbiamo bisogno per scrivere funzioni che hanno un elenco di argomenti di lunghezza variabile. L'header `<stdarg.h>` dichiara un tipo (`va_list`) e definisce diverse macro. Nel C89, ci sono tre macro, chiamate `va_start`, `va_arg` e `va_end`, che possono essere pensate come funzioni aventi i prototipi sopra elencati. Il C99 aggiunge una macro parametrica chiamata `va_copy`.

Per vedere come funzionano queste macro, le useremo per scrivere una funzione chiamata `max_int` che trova il massimo tra un numero *qualsiasi* di argomenti interi. La funzione potrà essere chiamata in questo modo:

```
max_int(3, 10, 30, 20)
```

Il primo argomento specifica quanti argomenti aggiuntivi seguiranno. Questa chiamata alla `max_int` restituisce il valore 30 (il maggiore tra i numeri 10, 30 e 20). Ecco la definizione della funzione:

```

int max_int(int n, ...) /* n deve essere almeno pari a 1 */
{
 va_list ap;
 int i, current, largest;

 va_start(ap, n);
 largest = va_arg(ap, int);

 for (i = 1; i < n; i++) {
 current = va_arg(ap, int);
 if (current > largest)
 largest = current;
 }

 va_end(ap);
 return largest;
}

```

Il simbolo ... presente nell'elenco dei parametri (conosciuto come **ellissi**) indica che il parametro n è seguito da un numero variabile di argomenti aggiuntivi.

Il corpo della funzione max\_int inizia con la dichiarazione di una variabile di tipo va\_list:

va\_list ap;

Dichiarare questa variabile è obbligatorio per max\_int affinché possa essere in grado di accedere agli argomenti che seguono n.

**va\_start** L'istruzione

va\_start(ap, n);

indica dove inizia la parte a lunghezza variabile dell'elenco degli argomenti (in questo caso dopo n). Una funzione con un numero variabile di argomenti deve avere almeno un parametro "normale". L'ellissi va sempre alla fine della lista dei parametri, dopo l'ultimo parametro ordinario.

**va\_arg** L'istruzione

largest = va\_arg(ap, int);

carica il suo secondo argomento (quello dopo la n), lo assegna alla variabile largest e automaticamente avanza a quello successivo. La parola int indica che ci aspettiamo che il secondo argomento della funzione max\_int sia di tipo int. L'istruzione

current = va\_arg(ap, int);

carica i rimanenti argomenti della funzione uno alla volta così come viene fatto all'interno del ciclo.



Non dimenticate che la macro va\_arg avanza sempre all'argomento successivo dopo aver caricato quello corrente. A causa di questa proprietà, non avremmo potuto scrivere il ciclo della max\_int nel modo seguente:

```

for (i = 1; i < n; i++)
 if (va_arg(ap, int) > largest) *** SBAGLIATO ***
 largest = va_arg(ap, int);

```

`va_end` L'istruzione

`va_end(ap);`

è necessaria per "fare pulizia" prima che la funzione termini (oppure, invece di terminare, la funzione potrebbe chiamare la `va_start` e attraversare nuovamente la lista degli argomenti).

`va_copy` La macro `va_copy` copia `src` (un valore `va_list`) all'interno di `dest` (anch'esso di tipo `va_list`). L'utilità di `va_copy` risiede nel fatto che delle chiamate multiple della `va_arg` possono essere fatte usando `src` prima che venga copiata all'interno di `dest` e quindi elaborando alcuni degli argomenti. Chiamare la `va_copy` permette a una funzione di ricordare il punto in cui si trova all'interno dell'elenco degli argomenti in modo da potervi ritornare successivamente per riesaminare un argomento (ed eventualmente anche gli argomenti che lo seguono).

Ogni chiamata alla `va_start` o alla `va_copy` deve essere associata a una chiamata alla `va_end`, e questa deve comparire all'interno della stessa funzione. Tutte le chiamate alla `va_arg` devono trovarsi tra la chiamata alla `va_start` (o `va_copy`) e la corrispondente chiamata `va_end`.



Quando una funzione con un elenco variabile di argomenti viene invocata, il compilatore esegue le promozioni di default degli argomenti [promozioni di default degli argomenti > 9.3] su tutti gli argomenti che si associano all'ellissi. In particolare, gli argomenti `char` e `short` vengono promossi al tipo `int`, mentre i valori `float` vengono promossi al tipo `double`. Di conseguenza non ha senso passare alla `va_arg` argomenti `char`, `short` o `float`, dato che dopo la promozione non possederanno nessuno di questi tipi.

## Chiamare una funzione con un elenco variabile di argomenti

Chiamare una funzione con un elenco variabile di argomenti è una cosa intrinsecamente rischiosa. Fin dal Capitolo 3 abbiamo visto come possa essere pericoloso passare degli argomenti sbagliati alle funzioni `printf` e `scanf`. Le altre funzioni con un elenco variabile di argomenti sono ugualmente sensibili. La difficoltà principale è che una funzione con un elenco variabile di argomenti non ha modo di determinare il loro numero e il loro tipo. Queste informazioni devono essere passate alla funzione e/o assunte dalla funzione. La funzione `max_int` si affida al primo argomento per specificare quanti argomenti aggiuntivi seguono. La funzione assume che gli argomenti siano di tipo `int`. Funzioni come la `printf` e la `scanf` si affidano alla stringa di formato che descrive il numero di argomenti aggiuntivi e il tipo di ognuno di questi.

Un altro problema ha a che fare con il passaggio del valore `NULL` come argomento. Di solito `NULL` viene definito in modo che rappresenti il valore 0. Quando 0 viene passato a una funzione con un elenco variabile di argomenti, il compilatore assume che questo rappresenti un intero. Il compilatore non ha modo di sapere che vorremmo che questo valore rappresenti un puntatore nullo. La soluzione è quella di aggiungere un cast, scrivendo `(void *) NULL` o `(void *) 0` invece di `NULL` (leggete la Sezione D&R alla fine del Capitolo 17 per una discussione approfondita su questo punto).

## Le funzioni v...printf

```

int vfprintf(FILE * restrict stream,
 const char * restrict format,
 va_list arg); da <stdio.h>
int vprintf(const char * restrict format,
 va_list arg); da <stdio.h>
int vsnprintf(char * restrict s, size_t n,
 const char * restrict format,
 va_list arg); da <stdio.h>
int vsprintf(char * restrict s,
 const char * restrict format,
 va_list arg); da <stdio.h>

```

`vfprintf`, `vprintf`, e `vsprintf` ("le funzioni v...printf") appartengono a `<stdio.h>`. Ne discutiamo in questa sezione perché vengono invariabilmente usate in congiunzione con le macro presenti in `<stdarg.h>`. Il C99 aggiunge la funzione `vsnprintf`.

C99

Le funzioni v...printf sono strettamente legate alle funzioni `fprintf`, `printf` e `sprintf`. A differenza di queste funzioni però, le v...printf possiedono un numero prefisso di argomenti. Ognuna di queste funzioni ha come ultimo argomento un valore `va_list`, il che implica che verrà chiamata da una funzione con un elenco variabile di argomenti. In pratica, le funzioni v...printf vengono usate principalmente per scrivere delle funzioni "wrapper" che accettino un numero variabile di argomenti che vengono passati a una funzione v...printf. Come esempio, supponiamo di lavorare su un programma che abbia bisogno di visualizzare di volta in volta dei messaggi di errore. Vorremmo che ogni messaggio iniziasse con una forma predeterminata:

`** Error n:`

dove il valore `n` è pari a 1 per il primo messaggio di errore e viene incrementato di un'unità per ogni errore seguente. Per rendere facile la produzione dei messaggi di errore scriveremo una funzione chiamata `errorf` che è simile alla `printf`, ma che aggiunge la stringa `** Error n:` all'inizio del suo output e scrive su `stderr` invece che su `stdout`. Faremo in modo che la `errorf` chiama la `vfprintf` per effettuare la maggior parte dell'output. Ecco come si presenta la funzione:

```

int errorf(const char *format, ...)
{
 static int num_errors = 0;
 int n;
 va_list ap;

 num_errors++;
 fprintf(stderr, "** Error %d: ", num_errors);
 va_start(ap, format);
 n = vfprintf(stderr, format, ap);
 va_end(ap);
 fprintf(stderr, "\n");
 return n;
}

```

La funzione wrapper (`errorf` nel nostro esempio) ha la responsabilità di chiamare la `va_start` prima di chiamare la funzione `v...printf` e di chiamare la `va_end` dopo che quest'ultima è terminata. La funzione wrapper può chiamare la `va_arg` una o più volte prima di chiamare la funzione `v...printf`.

**vsprintf**  
La funzione `vsprintf` è stata aggiunta alla versione C99 di `<stdio.h>` e corrisponde alla `sprintf` (discussa nella Sezione 22.8) che a sua volta è una funzione del C99.

## C99 Le funzioni v...scanf

```
int vfscanf(FILE * restrict stream,
 const char * restrict format,
 va_list arg); da <stdio.h>
int vscanf(const char * restrict format,
 va_list arg); da <stdio.h>
int vsscanf(const char * restrict s,
 const char * restrict format,
 va_list arg); da <stdio.h>
```

Il C99 aggiunge un insieme di "funzioni v...scanf" all'header `<stdio.h>`. Le funzioni `vfscanf`, `vscanf` e `vsscanf` sono rispettivamente equivalenti alle funzioni `fscanf`, `scanf` e `sscanf`, ma a differenza di queste ultime possiedono un parametro `va_list` attraverso il quale può essere passato un elenco variabile di argomenti. Come le funzioni `v...printf`, ogni funzione `v...scanf` è pensata per essere chiamata da una funzione wrapper che accetti un numero variabile di argomenti, i quali vengono poi passati alla funzione `v...scanf` stessa. La funzione wrapper ha la responsabilità di chiamare la `va_start` prima di chiamare la funzione `v...scanf` e di chiamare la `va_end` dopo che la quest'ultima è terminata.

## 26.2 L'header `<stdlib.h>`: utilità generali

L'header `<stdlib.h>` funge da raccoglitore per tutte quelle funzioni che non appartengono a nessun altro header. Le funzioni presenti in `<stdlib.h>` ricadono all'interno di otto gruppi:

- funzioni per le conversioni numeriche;
- funzioni per la generazione di sequenze pseudo casuali;
- funzioni per la gestione della memoria;
- comunicazioni con l'ambiente;
- utilità per la ricerca e per l'ordinamento;
- funzioni per l'aritmetica intera;
- funzioni di conversione tra caratteri multibyte e wide character;
- funzioni di conversione tra stringhe multibyte e stringhe wide.

Ci concentreremo a turno su ognuno di questi gruppi con tre eccezioni: le funzioni per la gestione della memoria, le funzioni di conversione tra i caratteri multibyte e i wide character, e le funzioni di conversione tra le stringhe multibyte e le stringhe wide.

Le funzioni per la gestione della memoria (`malloc`, `calloc`, `realloc` e `free`) permettono a un programma di allocare un blocco di memoria e successivamente rilasciarlo o modificare la sua dimensione. Il Capitolo 17 descrive tutte e quattro le funzioni con un certo dettaglio.

Le funzioni di conversione tra i caratteri multibyte e i wide character vengono usate per convertire un carattere multibyte in un wide character e viceversa. Le funzioni per la conversione delle stringhe multibyte e delle stringhe wide eseguono delle conversioni simili. Entrambi i gruppi di funzioni sono discussi nella Sezione 25.2.

## Le funzioni per le conversioni numeriche

```
double atof(const char *nptr),
int atoi(const char *nptr),
long int atol(const char *nptr),
long long int atoll(const char *nptr);

double strtod(const char * restrict nptr,
 char ** restrict endptr),
float strtof(const char * restrict nptr,
 char ** restrict endptr),
long double strtold(const char * restrict nptr,
 char ** restrict endptr),
long int strtol(const char * restrict nptr,
 char ** restrict endptr, int base),
long long int strtoll(const char * restrict nptr,
 char ** restrict endptr,
 int base),
unsigned long int strtoul(
 const char * restrict nptr,
 char ** restrict endptr, int base),
unsigned long long int strtoull(
 const char * restrict nptr,
 char ** restrict endptr, int base);
```

Le funzioni per le conversioni numeriche (o “funzioni per la conversione delle stringhe” come vengono chiamate nel C89) convertono delle stringhe contenenti dei numeri sotto forma di caratteri nei loro valori equivalenti. Tre di queste funzioni sono piuttosto vecchie, tre sono state aggiunte quando è stato creato il C89 e altre cinque sono state aggiunte dal C99.

C99

Tutte le funzioni per le conversioni numeriche (sia nuove che vecchie) funzionano più o meno allo stesso modo. Ogni funzione cerca di convertire una stringa (puntata dal parametro `nptr`) in un numero. Ogni funzione salta i caratteri di spazio bianco presenti all'inizio della stringa, tratta quelli successivi come facenti parte di un numero (che può eventualmente iniziare con un segno più o un segno meno), e si ferma al primo carattere che non può essere parte del numero stesso. Inoltre ogni funzione restituisce il valore zero se la conversione non può essere eseguita (la stringa è vuota,

o i caratteri successivi all'eventuale spazio bianco iniziale non seguono il formato che la funzione sta cercando).

**atof**  
**atoi**  
**atol**

**strtod**  
**strtol**  
**strtoul**

Le vecchie funzioni (atof, atoi e atol) convertono una stringa rispettivamente in un valore double, int o long int. Sfortunatamente queste funzioni sono prive di un modo per indicare quanta parte della stringa sia stata consumata durante la conversione. Inoltre queste funzioni non possiedono un modo per indicare che la conversione non è andata a buon fine (alcune implementazioni di queste funzioni possono modificare la variabile **errno** [variabile **errno** > 24.2] quando la conversione fallisce, ma questo non è garantito). Le funzioni C89 (strtod, strtol e strtoul) sono più sofisticate. Innanzitutto indicano dove si è fermata la conversione modificando la variabile puntata da **endptr** (il secondo argomento può essere un puntatore nullo se non siamo interessati a dove è terminata la conversione). Per controllare se una funzione è in grado di consumare un'intera stringa, possiamo semplicemente vedere se questa variabile punta a un carattere null. Se la conversione non è potuta avvenire, alla variabile puntata da **endptr** viene assegnato il valore **nptr** (se **endptr** non è un puntatore nullo). Inoltre le funzioni strtol e strtoul possiedono un argomento **base** che specifica la base del numero che deve essere convertito. Sono supportate tutte le basi dalla 2 alla 36 (incluso).

**atoll**  
**strtod**  
**strtold**  
**strtoll**  
**strtoull**

Oltre a essere più versatili rispetto alle funzioni più vecchie, la strtod, la strtol e la strtoul sono migliori nel rilevamento degli errori. Ogni funzione salva il valore ERANGE nella variabile **errno** nel caso in cui una conversione producesse un valore che va al di fuori dell'intervallo del valore restituito. In aggiunta la funzione strtod restituisce il valore **HUGE\_VAL** [macro **HUGE\_VAL** > 23.3] (con il segno più o con il segno meno), mentre le funzioni strtol e strtoul restituiscono il più piccolo o il più grande valore per i loro rispettivi tipi restituiti (la funzione strtol restituisce il valore **LONG\_MIN** o il valore **LONG\_MAX**, mentre la strtoul restituisce **ULONG\_MAX** [macro di <limits.h> > 23.2]). Il C99 aggiunge le funzioni atoll, strtod, strtold, strtoll e strtoull. La funzione atoll è uguale alla atol ma a differenza di quest'ultima converte una stringa in un valore long long int. Le funzioni strtod e strtold sono uguali alla strtod, ma a differenza di questa convertono rispettivamente in un valore float e long double. La funzione strtoll è uguale alla strtol ma, a differenza di questa, converte una stringa in un valore unsigned long long int. Il C99 ha apportato una piccola modifica anche alle funzioni di conversione numerica a virgola mobile. Le stringhe passate alla strtod (e anche alle sue nuove cugine strtod e strtold) possono contenere: un numero a virgola mobile esadecimale, infinito o NaN.



## PROGRAMMA

### Testare le funzioni di conversione numerica

Il seguente programma converte una stringa nella forma numerica applicando ognuna delle sei funzioni di conversione numerica che esistono nel C89. Dopo aver chiamato le funzioni, il programma mostra anche se ogni conversione ha prodotto un risultato valido e se è stata in grado di consumare l'intera stringa. Il programma ottiene la stringa di input dalla riga di comando.

```
tnumconv.c /* Testa le funzioni di conversione numerica del C89 */

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

```

#define CHK_VALID printf(" %s %s\n",
 errno != ERANGE ? "Yes" : "No ", \
 *ptr == '\0' ? "Yes" : "No")

int main(int argc, char *argv[])
{
 char *ptr;

 if (argc != 2) {
 printf("usage: tnumconv string\n");
 exit(EXIT_FAILURE);
 }

 printf("Function Return Value\n");
 printf("-----\n");
 printf("atof %g\n", atof(argv[1]));
 printf("atoi %d\n", atoi(argv[1]));
 printf("atol %ld\n\n", atol(argv[1]));

 printf("Function Return Value Valid? "
 "String Consumed?\n"
 "----- ----- ----- -----"
 "\n");
 printf("-----\n");

 errno = 0;
 printf("strtod %-12g", strtod(argv[1], &ptr));
 CHK_VALID;

 errno = 0;
 printf("strtol %-12ld", strtol(argv[1], &ptr, 10));
 CHK_VALID;

 errno = 0;
 printf("strtoul %-12lu", strtoul(argv[1], &ptr, 10));
 CHK_VALID;

 return 0;
}

```

Se l'argomento della riga di comando fosse 3000000000, l'output del programma potrebbe avere il seguente aspetto:

| Function | Return Value | Valid? | String Consumed? |
|----------|--------------|--------|------------------|
| <hr/>    |              |        |                  |
| atof     | 3e+09        |        |                  |
| atoi     | 2147483647   |        |                  |
| atol     | 2147483647   |        |                  |
| Function | Return Value | Valid? | String Consumed? |
| <hr/>    |              |        |                  |
| strtod   | 3e+09        | Yes    | Yes              |
| strtol   | 2147483647   | No     | Yes              |
| strtoul  | 3000000000   | Yes    | Yes              |

Su molte macchine il numero 3000000000 è troppo grande per essere rappresentato da un intero di tipo long, sebbene sia valido come unsigned long. Le funzioni atoi e atol non hanno modo di indicare che il numero rappresentato dal loro argomento è fuori dall'intervallo. Nell'output mostrato, queste funzioni hanno restituito il valore 2147483647 (il più grande intero long), tuttavia lo standard C non garantisce questo comportamento. La funzione strtoul ha eseguito la conversione correttamente, la strtol ha restituito il valore 2147483647 (lo standard richiede che la funzione restituiscia il più grande intero long) e ha salvato il valore ERANGE nella variabile errno.

Se l'argomento della riga di comando fosse 123.456, l'output sarebbe

**Function      Return Value**

| atof | 123.456 |
|------|---------|
| atoi | 123     |
| atol | 123     |

**Function      Return Value      Valid?      String Consumed?**

| strtod  | 123.456 | Yes | Yes |
|---------|---------|-----|-----|
| strtol  | 123     | Yes | No  |
| strtoul | 123     | Yes | No  |

Tutte e sei le funzioni hanno trattato questa stringa come un numero valido, sebbene le funzioni intere si siano fermate al punto decimale. Le funzioni strtol e strtoul sono state in grado di segnalare che non hanno consumato completamente la stringa. Se l'argomento della riga di comando fosse foo, l'output sarebbe

**Function      Return Value**

| atof | 0 |
|------|---|
| atoi | 0 |
| atol | 0 |

**Function      Return Value      Valid?      String Consumed?**

| strtod  | 0 | Yes | No |
|---------|---|-----|----|
| strtol  | 0 | Yes | No |
| strtoul | 0 | Yes | No |

Tutte le funzioni hanno individuato la lettera f e hanno immediatamente restituito il valore zero. Le funzioni str... non hanno modificato la variabile errno, ma possiamo capire che qualcosa è andato storto dal fatto che le funzioni non hanno consumato la stringa.

## Funzioni per la generazione di sequenze pseudo casuali

```
int rand(void);
void srand(unsigned int seed);
```

Le funzioni `rand` e `srand` supportano la generazione di numeri pseudo casuali. Queste funzioni sono utili nei programmi di simulazione e nei programmi di gioco (per esempio per simulare il lancio dei dadi o la distribuzione delle carte in un gioco).

- `rand` Ogni volta che viene chiamata, la funzione `rand` restituisce un numero compreso tra 0 e `RAND_MAX` (una macro definita in `<stdlib.h>`). I numeri restituiti da questa funzione non sono veramente casuali, ma vengono generati a partire da un "seme" (*seed*). All'osservatore casuale, però, sembra che la funzione `rand` produca effettivamente una sequenza di numeri non correlati tra loro.
- `srand` Chiamare la funzione `srand` fornisce il seme per la funzione `rand`. Se la `rand` viene chiamata prima della `srand`, il valore del seme viene assunto pari a 1. Ogni seme determina una particolare sequenza di numeri pseudo casuali, la `srand` ci permette di selezionare quale sequenza vogliamo.

Un programma che usasse sempre lo stesso seme otterrebbe sempre la stessa sequenza di numeri dalla funzione `rand`. A volte questa proprietà può essere utile: il programma si comporta esattamente nello stesso modo ogni volta che viene eseguito rendendolo più semplice da testare. Tuttavia, di solito vorremmo che la funzione `rand` producesse una sequenza *diversa* ogni volta che il programma viene eseguito (un programma per il gioco del poker che distribuisse sempre le stesse carte probabilmente non diventerebbe molto popolare). Il modo più semplice per rendere casuale il valore usato per il seme è quello di chiamare la funzione `time` [funzione `time` > 26.3] che restituisce un numero che codifica la data e l'ora corrente. Passare il valore restituito dalla funzione `time` alla `srand` fa in modo che la funzione `rand` vari da un'esecuzione all'altra. Per avere degli esempi di questa tecnica leggete i programmi `guess.c` e `guess2.c` (Sezione 10.2).

PROGRAMMA

## Testare le funzioni per la generazione di sequenze pseudo casuali

Il programma seguente visualizza i primi cinque valori restituiti dalla funzione `rand` e poi permette all'utente di scegliere un nuovo valore per il seme. Il processo si ripete fino a quando l'utente non immette lo zero come valore per il seme.

```
/* Testa le funzioni per la generazione di sequenze pseudo casuali */

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

int main(void)
{
 int i, seed;

 printf("This program displays the first five values of "
 "'rand.\n");

 for (;;) {
 for (i = 0; i < 5; i++)
 printf("%d ", rand());
 printf("\n\n");
 printf("Enter new seed value (0 to terminate): ");
 }
}
```

```

 scanf("%d", &seed);
 if (seed == 0)
 break;
 srand(seed);
 }
 return 0;
}

```

Ecco come potrebbe presentarsi una sessione del programma:

```

This program displays the first five values of rand.
1804289383 846930886 1681692777 1714636915 1957747793

```

```

Enter new seed value (0 to terminate): 100
677741240 611911301 516687479 1039653884 807009856

```

```

Enter new seed value (0 to terminate): 1
1804289383 846930886 1681692777 1714636915 1957747793

```

```

Enter new seed value (0 to terminate): 0

```

Ci sono molti modi per scrivere la funzione `rand` e quindi non c'è garanzia che ogni versione di questa funzione generi i numeri mostrati qui. Osservate che scegliere il valore 1 come seme restituisce la stessa sequenza di numeri che si otterrebbero senza specificare nessun seme.

## Comunicazione con l'ambiente

```

void abort(void);
int atexit(void (*func)(void));
void exit(int status);
void _Exit(int status);
char *getenv(const char *name);
int system(const char *string);

```

Le funzioni di questo gruppo forniscono una semplice interfaccia verso il sistema operativo, permettendo ai programmi di: (1) terminare sia normalmente che in modo anormale restituendo al sistema operativo un codice di stato, (2) prelevare informazioni dall'ambiente dell'utente, e (3) eseguire i comandi del sistema operativo. Una delle funzioni, la `_Exit`, è un'aggiunta del C99.



**exit** Effettuare la chiamata `exit(n)` in qualche punto del programma di solito è equivalente a eseguire l'istruzione `return n;` all'interno del `main`. In tal caso infatti il programma termina ed `n` viene restituito al sistema operativo come codice di stato. L'header `<stdlib.h>` definisce le macro `EXIT_FAILURE` ed `EXIT_SUCCESS` che possono essere usate come argomento della funzione `exit`. L'unico altro argomento portabile è lo 0, che possiede lo stesso significato di `EXIT_SUCCESS`. Restituire dei codici di stato diversi da questi è ammesso ma non è necessariamente portabile su tutti i sistemi operativi.



**atexit** Quando un programma termina, solitamente esegue dietro le quinte alcune operazioni finali. Queste operazioni includono lo svuotamento dei buffer di output contenenti-

ti dati non scritti, la chiusura degli stream aperti e l'eliminazione dei file temporanei. Potremmo volere che il programma esegua anche altre azioni prima di terminare. La funzione atexit ci permette di "registrare" una funzione che venga chiamata all'atto della conclusione del programma. Per registrare una funzione chiamata cleanup, per esempio, possiamo chiamare la atexit nel seguente modo:

```
atexit(cleanup);
```

Quando passiamo alla atexit un puntatore a funzione, questa salva tale puntatore per riferimenti futuri. Se successivamente il programma termina normalmente (attraverso una chiamata alle funzioni exit o un'istruzione return nella funzione main), qualsiasi funzione registrata con la atexit verrà chiamata automaticamente (se sono state registrate due o più funzioni, queste vengono chiamate nell'ordine inverso a quello di registrazione).

\_Exit Le funzione \_Exit è simile alla exit. Tuttavia la \_Exit non chiama le funzioni che sono state registrate con la atexit e non chiama nemmeno gli handler di segnale che sono stati precedentemente passati alla funzione signal [**funzione signal > 24.3**]. Inoltre la \_Exit non necessariamente svuota i buffer di output, chiude gli stream o cancella i file temporanei (se queste azioni vengano eseguite dipende dall'implementazione).

abort La funzione abort è simile alla exit, ma chiamarla fa sì che il programma termini in modo anomale. Le funzioni registrate con la funzione atexit non vengono chiamate. A seconda dell'implementazione, potrebbe succedere che i buffer di output contenenti dati non scritti non vengano svuotati, che gli stream non vengano chiusi e che i file temporanei non vengano cancellati. La funzione abort restituisce un codice di stato definito dall'implementazione che indica una conclusione senza successo.



getenv Molti sistemi operativi forniscono un "ambiente": un insieme di stringhe che descrivono le caratteristiche dell'utente. Queste stringhe tipicamente includono il percorso nel quale si deve cercare quando l'utente esegue un programma, il tipo del terminale dell'utente (nel caso dei sistemi multiutente) e così via. Per esempio, un percorso di ricerca di UNIX può somigliare al seguente:

```
PATH=/usr/local/bin:/bin:/usr/bin:..
```

La funzione getenv fornisce un accesso a tutte le stringhe dell'ambiente associato all'utente. Per esempio, per trovare il valore corrente della stringa PATH possiamo scrivere

```
char *p = getenv("PATH");
```

ora p punta alla stringa "/usr/local/bin:/bin:/usr/bin:..". Fate attenzione con la getenv perché restituisce un puntatore a una stringa allocata staticamente che potrebbe essere modificata da una successiva chiamata alla stessa funzione.

system La funzione system permette a un programma C di eseguire un altro programma (eventualmente un comando del sistema operativo). L'argomento della funzione system è una stringa contenente un comando, simile a quello che avremmo immesso nel prompt del sistema operativo. Per esempio, supponete di scrivere un programma che abbia bisogno di un elenco dei file contenuti nella directory corrente. Un programma UNIX chiamerebbe la funzione system nel modo seguente:

```
system("ls >myfiles");
```

Questa chiamata invoca il comando UNIX ls e chiede a questo di scrivere un elenco per la directory corrente in un file chiamato myfiles.

Il valore restituito dalla `system` è definito dall'implementazione. Tipicamente questa funzione restituisce il codice di stato del programma che si è voluto far eseguire. Analizzare tale valore ci permette di controllare se il programma ha funzionato correttamente. Chiamare la funzione `system` con un puntatore nullo possiede un significato speciale: la funzione restituisce un valore diverso da zero se è disponibile un processore di comandi.

## Utilità per la ricerca e l'ordinamento

```
void *bsearch(const void *key, const void *base,
 size_t nmemb, size_t size,
 int (*compar)(const void *,
 const void *));
void qsort(void *base, size_t nmemb, size_t size,
 int (*compar)(const void *, const void *));
```

### bsearch

La funzione `bsearch` ricerca un particolare valore (la "chiave") in un vettore ordinato. Quando la `bsearch` viene chiamata, il parametro `key` punta alla chiave, `base` punta al vettore, `nmemb` punta al numero di elementi presenti nel vettore, `size` è la dimensione di ogni elemento (espressa in byte) e `compar` è un puntatore a una funzione di confronto. La funzione di confronto è simile a quella richiesta dalla `qsort`: quando le vengono passati i puntatori alla chiave e un vettore di elementi (in quell'ordine). La funzione deve restituire un valore intero negativo, pari a zero o positivo a seconda che la chiave sia minore, uguale o maggiore dell'elemento del vettore. La funzione `bsearch` restituisce un puntatore a un elemento che combacia con la chiave. Se la funzione non trova nessuna corrispondenza allora restituisce un puntatore nullo.

Sebbene lo standard C non lo richieda, normalmente la `bsearch` usa un algoritmo di ricerca binaria per effettuare le ricerche all'interno del vettore. Per prima cosa la funzione confronta la chiave con l'elemento posto nel mezzo del vettore, se c'è una corrispondenza la funzione termina. Se la chiave è minore dell'elemento di mezzo, la funzione circoscrive la sua ricerca nella prima metà del vettore. Se la chiave è maggiore, allora la funzione effettua le ricerche solo nella seconda metà del vettore. La `bsearch` ripete questa strategia fino a quando trova la chiave o non ha più elementi nei quali cercare. Grazie a questa tecnica la `bsearch` è piuttosto veloce: effettuare una ricerca in un vettore di 1000 elementi richiede al più 10 confronti, mentre cercare all'interno di un vettore di 1.000.000 elementi non richiede più di 20 confronti.

### qsort

La Sezione 17.7 tratta la funzione `qsort`, la quale può ordinare qualsiasi vettore. La `bsearch` funziona solo con i vettori ordinati, ma possiamo sempre usare la `qsort` per ordinare un vettore prima di chiederle di effettuare la ricerca.

### PROGRAMMA

## Determinare le miglia aeree

Il nostro prossimo programma calcola le miglia aeree che intercorrono tra New York e varie città internazionali. Per prima cosa il nostro programma chiede all'utente di

immettere il nome di una città e poi visualizza la distanza in miglia che intercorre quest'ultima e la città di New York:

```
Enter city name: Shanghai
Shanghai is 7371 miles from New York City.
```

Il programma manterrà delle coppie città-miglia in un vettore. Utilizzando la funzione `bsearch` per cercare il nome della città all'interno del vettore, il programma potrà trovare facilmente la distanza corrispondente (le distanze in miglia sono tratte da [Infoplease.com](http://www.infoplease.com)).

```

/* Determinare le miglia aeree tra New York ed altre città */

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

struct city_info {
 char *city;
 int miles;
};

int compare_cities(const void *key_ptr,
 const void *element_ptr);

int main(void)
{
 char city_name[81];
 struct city_info *ptr;
 const struct city_info mileage[] =
 {{"Berlin", 3965}, {"Buenos Aires", 5297},
 {"Cairo", 5602}, {"Calcutta", 7918},
 {"Cape Town", 7764}, {"Caracas", 2132},
 {"Chicago", 713}, {"Hong Kong", 8054},
 {"Honolulu", 4964}, {"Istanbul", 4975},
 {"Lisbon", 3364}, {"London", 3458},
 {"Los Angeles", 2451}, {"Manila", 8498},
 {"Mexico City", 2094}, {"Montreal", 320},
 {"Moscow", 4665}, {"Paris", 3624},
 {"Rio de Janeiro", 4817}, {"Rome", 4281},
 {"San Francisco", 2571}, {"Shanghai", 7371},
 {"Stockholm", 3924}, {"Sydney", 9933},
 {"Tokyo", 6740}, {"Warsaw", 4344},
 {"Washington", 205}};

 printf("Enter city name: ");
 scanf("%80[^\\n]", city_name);
 ptr = bsearch(city_name, mileage,
 sizeof(mileage) / sizeof(mileage[0]),
 sizeof(mileage[0]), compare_cities);
}

```

```

if (ptr != NULL)
 printf("%s is %d miles from New York City.\n",
 city_name, ptr->miles);
else
 printf("%s wasn't found.\n", city_name);

return 0;
}

int compare_cities(const void *key_ptr,
 const void *element_ptr)
{
 return strcmp((char *) key_ptr,
 ((struct city_info *) element_ptr)->city);
}

```

## Funzioni per l'aritmetica intera

```

int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);

div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer,
 long long int denom);

```

**abs** La funzione `abs` restituisce il valore assoluto di un valore `int`, mentre la funzione `labs` restituisce il valore assoluto di un valore `long int`.

**div** La funzione `div` divide il suo primo argomento per il secondo restituendo un valore `div_t`. Il tipo `div_t` è una struttura che contiene sia un membro quoiente (chiamato `quot`) che un membro resto (chiamato `rem`). Per esempio, se `ans` è una variabile di tipo `div_t`, possiamo scrivere

```

ans = div(5, 2);
printf("Quotient: %d Remainder: %d\n", ans.quot, ans.rem);

```

La funzione `ldiv` è simile ma lavora con interi di tipo `long`. Questa restituisce una struttura `ldiv_t` che a sua volta possiede i membri `quot` e `rem` (i tipi `div_t` e `ldiv_t` vengono dichiarati in `<stdlib.h>`).

**llabs** Il C99 fornisce due funzioni aggiuntive. La funzione `llabs` restituisce il valore assoluto di un valore `long long int`. La funzione `lldiv` è simile alle funzioni `div` e `ldiv` ma a differenza di queste divide due valori `long long int` e restituisce una struttura `lldiv_t` (anche il tipo `lldiv_t` è stato aggiunto dal C99).

## 26.3 L'header `<time.h>`: data e ora

L'header `<time.h>` fornisce delle funzioni per determinare l'ora (includendo la data), eseguire dell'aritmetica sulle ore e formattarle per la visualizzazione. Prima di esplorare le funzioni di questo header, è utile conoscere alcuni concetti fondamentali.

rare queste funzioni, però, dobbiamo discutere di come le ore vengono memorizzate. L'header `<time.h>` fornisce tre tipi, ognuno dei quali rappresentante un diverso modo di memorizzare un orario:

- `clock_t`: un valore orario misurato in "tick del clock";
- `time_t`: un formato compatto per la codifica dell'ora e dalla data (questo formato viene detto **calendar time**);
- `struct tm`: un'ora che è stata divisa in secondi, minuti, ore e così via. Un valore del tipo `struct tm` viene spesso chiamato **broken-down time**. La Tabella 26.1 illustra i membri della struttura `tm`. Tutti i membri sono di tipo `int`.

**Tabella 26.1** Membri della struttura `tm`

| Nome                  | Descrizione               | Valore minimo | Valore massimo  |
|-----------------------|---------------------------|---------------|-----------------|
| <code>tm_sec</code>   | Secondi dopo il minuto    | 0             | 61 <sup>†</sup> |
| <code>tm_min</code>   | Minuti dopo l'ora         | 0             | 59              |
| <code>tm_hour</code>  | Ore dopo la mezzanotte    | 0             | 23              |
| <code>tm_mday</code>  | Giorno del mese           | 1             | 31              |
| <code>tm_mon</code>   | Mesi a partire da gennaio | 0             | 11              |
| <code>tm_year</code>  | Anni dal 1900             | 0             | -               |
| <code>tm_wday</code>  | Giorni dalla domenica     | 0             | 6               |
| <code>tm_yday</code>  | Giorni dal 1 gennaio      | 0             | 365             |
| <code>tm_isdst</code> | Flag ora legale           | †             | †               |

<sup>†</sup>Permette un salto di due secondi. Nel C99 il valore massimo è 60.

†Il valore è positivo se è attiva l'ora legale, zero se non è attiva e negativo se questa informazione è sconosciuta.

Questi tipi vengono usati per scopi differenti. Un valore `clock_t` è utile per rappresentare una durata temporale, mentre i valori `time_t` e `struct tm` possono contenere un'intera data e ora. I valori `time_t` hanno una codifica molto spinta e per questo occupano poco spazio. I valori `struct tm` richiedono più spazio, ma spesso sono più facili da manipolare. Lo standard C asserisce che `clock_t` e `time_t` devono essere dei "tipi aritmetici", ma non specifica oltre. Non sappiamo nemmeno se i loro valori vengono memorizzati come numeri interi o in virgola mobile.

Ora siamo pronti per trattare le funzioni dell'header `<time.h>`, che rientrano in due gruppi: le funzioni di manipolazione delle ore e le funzioni di conversione delle ore.

## Funzioni per la manipolazione delle ore

```
clock_t clock(void);
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm *tmeptr);
time_t time(time_t *timer);
```

**clock** La funzione `clock` restituisce un valore `clock_t` rappresentante il tempo del processore usato dal programma a partire dall'inizio della sua esecuzione. Per convertire questo valore in secondi possiamo dividerlo per `CLOCKS_PER_SEC`, una macro definita in `<time.h>`.

Quando la funzione `clock` viene usata per determinare da quanto il programma sia in esecuzione è abitudine chiamarla due volte: una all'inizio della funzione `main` e una immediatamente prima che il programma termini:

```
#include <stdio.h>
#include <time.h>
int main(void)
{
 clock_t start_clock = clock();
 -
 printf("Processor time used: %g sec.\n",
 (clock() - start_clock) / (double) CLOCKS_PER_SEC);
 return 0;
}
```

La ragione della chiamata iniziale alla funzione `clock` è che il programma utilizzerà un po' del tempo del processore prima di raggiungere il `main` a causa del codice nascosto di *start-up*. Chiamare la funzione `clock` all'inizio del `main` determina quanto tempo richiede il codice di *start-up* in modo da poterlo sottrarre in un secondo momento.

Lo standard C89 dice solo che `clock_t` è un tipo aritmetico, mentre il tipo di `CLOCKS_PER_SEC` non viene specificato. Ne risulta quindi che il tipo dell'espressione

$$(clock() - start_clock) / CLOCKS_PER_SEC$$

può differire da un'implementazione all'altra, rendendola difficile da visualizzare usando la `printf`. Per risolvere questo problema il nostro esempio converte `CLOCKS_PER_SEC` al tipo `double`, forzando l'intera espressione al tipo `double`. Nel C99 la macro `CLOCKS_PER_SEC` è specificata essere di tipo `clock_t`, tuttavia `clock_t` è ancora un tipo definito dall'implementazione.

**C99**

**time** La funzione `time` restituisce la data e l'ora correnti. Se il suo argomento non è un puntatore nullo, la funzione salva anche l'ora nell'oggetto puntato da questo. La capacità di questa funzione di restituire l'ora in due modi diversi è una stranezza storica, tuttavia ci permette di scrivere sia

```
cur_time = time(NULL);
che
time(&cur_time);
```

dove `cur_time` è una variabile di tipo `time_t`.

**difftime** La funzione `difftime` restituisce la differenza espressa in secondi tra `time0` (un ora precedente) e `time1`. Quindi per calcolare il tempo di esecuzione corrente di un programma (non necessariamente il tempo del processore), possiamo usare il seguente codice:

```
#include <stdio.h>
#include <time.h>

int main(void)
{
 time_t start_time = time(NULL);
 ...
 printf("Running time: %g sec.\n",
 difftime(time(NULL), start_time));
 return 0;
}
```

**mktime** La funzione `mktime` converte un'ora di tipo broken-down (contenuta nella struttura puntata dall'argomento della funzione) in un'ora di tipo `calendar`, la quale viene restituita. Come side effect, la funzione modifica i membri della struttura attenendosi alle seguenti regole.

- La funzione `mktime` modifica tutti i membri il cui valore non rientra negli intervalli ammessi (vedi Tabella 26.1). Questa modifica può a sua volta richiedere delle modifiche agli altri membri. Se per esempio `tm_sec` è troppo grande, la funzione lo riduce all'intervallo appropriato (0-59) e aggiunge minuti extra al membro `tm_min`. Se ora è `tm_min` a essere troppo grande, la `mktime` lo riduce e aggiunge delle ore al membro `tm_hour`. Se necessario il processo continua aggiornando i membri `tm_mday`, `tm_mon` e `tm_year`.
- Dopo aver regolato gli altri membri della struttura (se necessario), la `mktime` imposta i membri `tm_wday` (giorno della settimana) e `tm_year` (giorno dell'anno) ai loro valori corrispondenti. Non c'è mai la necessità di inizializzare i valori di `tm_wday` e `tm_yday` prima della chiamata alla `mktime`, questa infatti ignora i loro valori originali.

L'abilità della funzione `mktime` di regolare i membri di una struttura `tm` la rende particolarmente utile per l'aritmetica associata alle date. Come esempio, usiamo la `mktime` per rispondere alla seguente domanda: se le olimpiadi del 2012 iniziano il 27 luglio e terminano 16 giorni più tardi, quale sarà la data di conclusione? Inizieremo memorizzando la data 27 luglio 2012 in una struttura `tm`:

```
struct tm t;
t.tm_mday = 27;
t.tm_mon = 6; /* Luglio */
t.tm_year = 112; /* 2012 */
```

Inizializziamo anche gli altri membri della struttura (a eccezione di `tm_wday` e `tm_yday`) per assicurarc che questi non contengano dei valori indefiniti che possano corrompere il risultato:

```
t.tm_sec = 0;
t.tm_min = 0;
t.tm_hour = 0;
t.tm_isdst = -1;
```

Successivamente sommiamo il valore 16 al membro `tm_day`:

```
t.tm_mday += 16;
```

Questa operazione lascia il valore 43 nel membro `tm_day`. Questo valore è al di fuori dell'intervallo accettabile per quel membro. La chiamata alla `mktime` riporterà i membri della struttura all'interno dei loro intervalli:

```
mktime(&t);
```

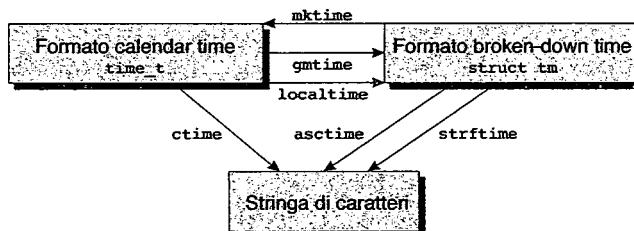
Scarteremo il valore restituito dalla `mktime`, dato che siamo interessati solamente all'effetto della funzione sulla variabile `t`. I membri di `t` ora possiedono i valori seguenti:

| Membro               | Valore | Significato           |
|----------------------|--------|-----------------------|
| <code>tm_mday</code> | 12     | 12                    |
| <code>tm_mon</code>  | 7      | Agosto                |
| <code>tm_year</code> | 112    | 2012                  |
| <code>tm_wday</code> | 0      | Domenica              |
| <code>tm_yday</code> | 224    | 225° giorno dell'anno |

## Funzioni per la conversione delle ore

```
char *asctime(const struct tm *timeptr),
char *ctime(const time_t *timer),
struct tm *gmtime(const time_t *timer),
struct tm *localtime(const time_t *timer),
size_t strftime(char * restrict s, size_t maxsize,
 const char * restrict format,
 const struct tm * restrict timeptr);
```

Le funzioni di conversione rendono possibile convertire ore di tipo calendar in ore di tipo broken-down. La figura seguente mostra come sono collegate queste funzioni:



La figura include la funzione `mktime` che viene classificata dallo standard C come una funzione di "manipolazione" invece che come una funzione di "conversione".

Le funzioni `gmtime` e `localtime` sono simili. Quando viene passato un puntatore a un'ora di tipo calendar, entrambe restituiscono un puntatore a una struttura contenente l'equivalente nel formato broken-down. La funzione `localtime` produce un'ora



locale, mentre il valore restituito dalla `gmtime` viene espresso in UTC (Tempo Coordinato Universale). Il valore restituito dalla `gmtime` e dalla `localtime` punta a una struttura allocata staticamente che può essere modificata da una successiva chiamata a una delle due funzioni.

`asctime` La funzione `asctime` (*ASCII time*) restituisce un puntatore a una stringa (terminata con `null`) della forma:

```
Sun Jun 3 17:48:34 2007\n
```

Tale stringa viene costruita a partire dall'ora broken-down puntata dal suo argomento.

`ctime` La funzione `ctime` restituisce un puntatore a una stringa che descrive l'ora locale. Se `cur_time` è una variabile di tipo `time_t`, la chiamata

```
ctime(&cur_time)
```

è equivalente alla chiamata

```
asctime(localtime(&cur_time))
```

Il valore restituito dalle funzioni `asctime` e `ctime` punta a una stringa allocata staticamente che può essere modificata da una successiva chiamata a una di queste funzioni.

`strftime` La funzione `strftime`, come la `asctime`, converte un'ora di tipo broken-down nel formato stringa. A differenza dalla `asctime`, però, questa funzione ci fornisce un ampio grado di controllo sulla formattazione dell'ora. Infatti la `strftime` ricorda la funzione `sprintf` [funzione `sprintf` > 22.8] in quanto scrive i caratteri in una stringa `s` (il primo argomento) in accordo con una stringa di formato (il terzo argomento). La stringa di formato può contenere normali caratteri (che vengono copiati all'interno di `s` senza essere modificati) assieme a specificatori di conversione elencati nella Tabella 26.2 (che vengono sostituiti dalle stringhe indicate). L'ultimo argomento punta a una struttura `tm` che viene usata come sorgente per le informazioni sulla data e sull'ora. Il secondo argomento è un limite al numero di caratteri che possono essere salvati nella stringa `s`.

La funzione `strftime`, diversamente dalle altre funzioni presenti in `<time.h>` è sensibile alla localizzazione corrente [localizzazione > 25.1]. Modificare la categoria `LC_TIME` può avere effetto sul comportamento degli specificatori di conversione. Gli esempi nella Tabella 26.2 sono strettamente legati alla localizzazione "C". In una localizzazione per la lingua tedesca `%A` può essere rimpiazzato da `Dienstag` invece che da `Tuesday`.

C99 Lo standard C99 enuncia le esatte stringhe di sostituzione nella localizzazione "C" per alcuni specificatori di conversione (lo standard C89 non entra in questo livello di dettaglio). La Tabella 26.3 elenca questi specificatori di conversione e le stringhe con le quali vengono rimpiazzati.

C99 Il C99 aggiunge un certo numero di specificatori di conversione per la funzione `strftime` (Tabella 26.2). Una delle ragioni per questi specificatori aggiuntivi è il desiderio di supportare lo standard ISO 8601.

**Tabella 26.2** Specifiche di conversione per la funzione strftime

| Nome | Descrizione                                                                     |
|------|---------------------------------------------------------------------------------|
| %a   | Nome del giorno della settimana abbreviato (per esempio Sun).                   |
| %A   | Nome del giorno della settimana intero (Sunday).                                |
| %b   | Nome del mese abbreviato (Jun).                                                 |
| %B   | Nome del mese completo (June).                                                  |
| %c   | Data e ora completa (Sun Jun 3 17:48:34 2007).                                  |
| %C†  | Anno diviso per 100 e troncato a intero (00-99).                                |
| %d   | Giorno del mese (01-31).                                                        |
| %D†  | Equivalente a %m/%d/%y.                                                         |
| %e†  | Giorno del mese (1-31), una singola cifra preceduta da uno spazio.              |
| %F†  | Equivalente a %Y-%m-%d.                                                         |
| %g†  | Ultime due cifre dell'anno basato sulla settimana dell'ISO 8601 (00-99).        |
| %G†  | Anno basato sulla settimana ISO 8601.                                           |
| %h†  | Equivalente a %b.                                                               |
| %H   | Ora su un orologio a 24 ore (00-23).                                            |
| %I   | Ora su un orologio a 12 ore (01-12).                                            |
| %j   | Giorno dell'anno (001-366).                                                     |
| %m   | Mese (01-12).                                                                   |
| %M   | Minuti (00-59).                                                                 |
| %n†  | Carattere new-line.                                                             |
| %p   | Indicatore AM/PM (AM o PM).                                                     |
| %r   | Orario su un orologio a 12 ore (05:48:34 PM).                                   |
| %R   | Equivalente a %H:%M.                                                            |
| %S   | Secondi (00-61), il massimo valore in C99 è 60.                                 |
| %t†  | Carattere tabulazione orizzontale.                                              |
| %T†  | Equivalente a %H:%M:%S.                                                         |
| %u†  | Giorno della settimana ISO 8601 (1-7). Il lunedì corrisponde a 1.               |
| %U   | Numero della settimana (00-53). La prima domenica è l'inizio della settimana 1. |
| %V†  | Numero della settimana ISO 8601 (01-53).                                        |
| %w   | Giorno della settimana (0-6). La domenica corrisponde allo 0.                   |
| %W   | Numero della settimana (00-53). Il primo lunedì è l'inizio della settimana 1.   |
| %x   | Data completa (per esempio 06/03/07).                                           |
| %X   | Ora completa (17:48:34).                                                        |
| %y   | Ultime due cifre dell'anno (00-99).                                             |
| %Y   | Anno.                                                                           |
| %z†  | Scostamento dall'UTC nel formato ISO 8601 (-0530 o +0200).                      |
| %Z   | Nome del fuso orario o abbreviazione (EST).                                     |
| %%   | %.                                                                              |

†solo C99.

**Tabella 26.3** Stringhe di sostituzione per le specifiche di conversione strftime nella localizzazione "C"

| Nome | Descrizione                                               |
|------|-----------------------------------------------------------|
| %a   | Primi tre caratteri di %A                                 |
| %A   | Una delle stringhe "Sunday", "Monday", ..., "Saturday"    |
| %b   | I primi tre caratteri di %B                               |
| %B   | Una delle stringhe "January", "February", ..., "December" |
| %c   | Equivalenti a "%a %b %e %T %Y"                            |
| %p   | "AM" o "PM"                                               |
| %r   | Equivalenti a "%I:%M:%S %p"                               |
| %x   | Equivalenti a "%m/%d/%y"                                  |
| %X   | Equivalenti a %T                                          |
| %z   | Definito dall'implementazione                             |

## ISO 8601

L'ISO 8601 è uno standard internazionale che descrive dei modi per rappresentare la data e l'ora. È stato pubblicato originariamente nel 1988 e successivamente aggiornato nel 2000 e nel 2004. Secondo questo standard la data e l'ora sono interamente numeriche (ovvero i mesi non sono rappresentati dai loro nomi) e le ore sono espresse usando l'orologio a 24 ore.

Per la data e per l'ora ci sono diversi formati ISO 8601, alcuni dei quali sono direttamente supportati nel C99 dagli specificatori di formato della funzione `strftime`. Il principale formato ISO 8601 per le date (`YYYY-MM-DD`) e il principale formato per le ore (`hh:mm:ss`) corrispondono rispettivamente agli specificatori di conversione `%F` e `%T`.

L'ISO 8601 possiede un sistema di numerazione delle settimane di un anno. Questo sistema è supportato dagli specificatori di conversione `%g`, `%G` e `%V`. Le settimane iniziano con il lunedì e la settimana 1 è quella contenente il primo giovedì dell'anno. Di conseguenza i primi giorni di gennaio (fino a tre) possono appartenere all'ultima settimana dell'anno precedente. Per esempio, considerate il calendario di gennaio dell'anno 2011:

### Gennaio 2011

| Lun | Mar | Mer | Gio | Ven | Sab | Dom | Anno | Settimana |
|-----|-----|-----|-----|-----|-----|-----|------|-----------|
|     |     |     |     |     | 1   | 2   | 2010 | 52        |
| 3   | 4   | 5   | 6   | 7   | 8   | 9   | 2011 | 1         |
| 10  | 11  | 12  | 13  | 14  | 15  | 16  | 2011 | 2         |
| 17  | 18  | 19  | 20  | 21  | 22  | 23  | 2011 | 3         |
| 24  | 25  | 26  | 27  | 28  | 29  | 30  | 2011 | 4         |
| 31  |     |     |     |     |     |     | 2011 | 5         |

Il 6 gennaio è il primo giovedì dell'anno, quindi quella dal 3 al 9 gennaio è la settimana 1. I giorni 1 e 2 gennaio appartengono all'ultima settimana (la settimana 52) dell'anno precedente. Per queste due date la funzione `strftime` sostituirà `%g` con 10, `%G` con 2010 e `%V` con 52. Osservate che gli ultimi giorni di dicembre a volte appartengono alla settimana 1 dell'anno seguente. Questo succede ogni volta che il 29, il 30 o il 31 dicembre corrisponde a un lunedì.

Lo specificatore di conversione `%z` corrisponde alla specifica del fuso orario: `-hhmm` significa che un fuso orario è `hh` ore e `mm` minuti indietro rispetto all'UTC. La stringa `+hhmm` indica l'ammontare di quanto un fuso orario è avanti rispetto all'UTC.

C99

Il C99 permette di usare i caratteri E e O per modificare il significato di certi specificatori di conversione della funzione strftime. Gli specificatori di conversione che iniziano con il modificatore E o il modificatore O fanno sì che per la sostituzione venga usato un formato alternativo dipendente dalla localizzazione. Se nella localizzazione corrente non esiste una rappresentazione alternativa, il modificatore non ha alcun effetto (nella localizzazione "C" la E e la O vengono ignorate). La Tabella 26.4 elenca tutti gli specificatori di conversione ai quali è permesso avere i modificatori E e O.

**Tabella 26.4** Specifiche di conversione modificate con E- e O- per la funzione strftime (solo C99)

| Nome | Descrizione                                                                                                                              |
|------|------------------------------------------------------------------------------------------------------------------------------------------|
| %Ec  | Rappresentazione alternativa della data e dell'ora.                                                                                      |
| %EC  | Nome dell'anno base (periodo) nella rappresentazione alternativa.                                                                        |
| %Ex  | Rappresentazione alternativa della data.                                                                                                 |
| %EX  | Rappresentazione alternativa dell'ora.                                                                                                   |
| %Ey  | Scostamento da %EC (solo l'anno) nella rappresentazione alternativa.                                                                     |
| %EY  | Rappresentazione alternativa completa dell'anno.                                                                                         |
| %od  | Giorno del mese usando simboli numerici alternativi (riempito con degli zeri iniziali se non c'è alcun simbolo alternativo per lo zero). |
| %oe  | Giorno del mese usando simboli numerici alternativi (riempito con degli spazi iniziali).                                                 |
| %OH  | Ora su un orologio a 24 ore usando simboli numerici alternativi.                                                                         |
| %OI  | Ora su un orologio a 12 ore usando simboli numerici alternativi.                                                                         |
| %Om  | Mese usando simboli numerici alternativi.                                                                                                |
| %OM  | Minuti usando simboli numerici alternativi.                                                                                              |
| %OS  | Secondi usando simboli numerici alternativi.                                                                                             |
| %ou  | Giorno della settimana ISO 8601 come numero in una rappresentazione alternativa, dove il lunedì corrisponde a 1.                         |
| %OU  | Numero della settimana usando simboli numerici alternativi.                                                                              |
| %OV  | Numero della settimana ISO 8601 usando simboli numerici alternativi.                                                                     |
| %ow  | Giorno della settimana come numero usando simboli numerici alternativi.                                                                  |
| %OW  | Giorno della settimana usando simboli numerici alternativi.                                                                              |
| %oy  | Ultime due cifre dell'anno usando simboli numerici alternativi.                                                                          |

## Visualizzare la data e l'ora

Supponiamo di voler scrivere un programma che visualizzi la data e l'ora correnti. Naturalmente il primo passo del programma è una chiamata alla funzione time per ottenere l'ora corrente nel formato calendar time. Il secondo passo è quello di convertire l'ora nel formato stringa e di stamparla. Il modo più semplice per effettuare il secondo passo è quello di chiamare la funzione ctime che restituisce un puntatore a una stringa contenente la data e l'ora, e poi passare questo puntatore alla funzione puts o alla printf.

Fino a qui va bene, ma se volessimo che il programma visualizzasse la data e l'ora in un modo particolare? Assumiamo di volere il seguente formato, dove 06 è il mese e 03 è il giorno del mese:

06-03-2007 5:48p

La funzione `ctime` utilizza sempre lo stesso formato per la data e l'ora, quindi non di alcun aiuto. La funzione `strftime` è migliore, usandola possiamo quasi raggiungere la rappresentazione voluta. Sfortunatamente la `strftime` non ci lascerà visualizzare un'ora su una cifra senza metterci uno zero davanti. Inoltre la funzione `strftime` utilizza AM e PM invece di a e p.

Quando la `strftime` non è sufficiente, abbiamo un'altra alternativa: convertire un'ora del tipo `calendar time` in una del tipo `broken-down` e poi estrarre le informazioni rilevanti dalla struttura `tm` e formattarle noi stessi usando la `printf` o una funzione simile. Possiamo usare anche la `strftime` per effettuare alcune formattazioni prima che le altre funzioni completino il lavoro.

Il programma seguente illustra le diverse opzioni. Visualizza la data e l'ora corrente in tre formati: quello usato dalla `ctime`, uno simile a quello che volevamo (creato usando la `strftime`) e quello che volevamo (creato usando la `printf`). La versione `ctime` è semplice da fare, la versione `strftime` è leggermente più complicata e la versione `printf` è la più difficile.

```
datetime.c /* Visualizza la data e l'ora corrente in tre formati */

#include <stdio.h>
#include <time.h>

int main(void)
{
 time_t current = time(NULL);
 struct tm *ptr;
 char date_time[21];
 int hour;
 char am_or_pm;

 /* Stampa la data e l'ora nel formato di default */
 puts(ctime(¤t));

 /* Stampa la data e l'ora usando la strftime per la formattazione */
 strftime(date_time, sizeof(date_time),
 "%m-%d-%Y %I:%M%p\n", localtime(¤t));
 puts(date_time);

 /* Stampa la data e l'ora usando la printf per la formattazione */
 ptr = localtime(¤t);
 hour = ptr->tm_hour;
 if (hour <= 11)
 am_or_pm = 'a';
 else {
 hour -= 12;
 am_or_pm = 'p';
 }
}
```

```

 }
 if (hour == 0)
 hour = 12;

 printf("%.2d-%.2d-%d %.2d:%.2d%c\n", ptr->tm_mon + 1,
 ptr->tm_mday, ptr->tm_year + 1900, hour,
 ptr->tm_min, am_or_pm);

 return 0;
}

```

L'output di `datetime.c` si presenterà in questo modo:

```

Sun Jun 3 17:48:34 2007
06-03-2007 05:48PM
06-03-2007 5:48p

```

## Domande & Risposte

**D:** Sebbene l'header `<stdlib.h>` fornisca un certo numero di funzioni convertono le stringhe in numeri, non sembra esserci nessuna funzione che converte i numeri in stringhe.

**R:** Alcune librerie del C forniscono delle funzioni come `itoa` che convertano numeri in stringhe. Usare queste funzioni non è una buona idea: non fanno parte del standard C e non sono portabili. Il modo migliore per eseguire questo tipo di conversioni è chiamare una funzione come la `sprintf` [funzione `sprintf` > 22.8] che scrive l'output formattato in una stringa:

```

char str[20];
int i;
-
sprintf(str, "%d", i); /* scrive i nella stringa str */

```

Non solo la `sprintf` è portabile, ma fornisce anche un buon grado di controllo sull'aspetto del numero.

\***D:** La descrizione della funzione `strtod` dice che il C99 permette all'argomento stringa di contenere un numero a virgola mobile esadecimale infinito o NaN. Qual è il formato di questi numeri? [p. 709]

**R:** Un numero a virgola mobile esadecimale inizia con `0x` o `0X`, seguito da uno o più cifre esadecimali (includendo eventualmente il carattere del punto decimali) e eventualmente un esponente binario (leggete la Sezione Domande & Risposte alla fine del Capitolo 7 per una discussione sulle costanti esadecimali a virgola mobile). I numeri hanno un formato simile, ma non identico). L'infinito è espresso come `INF` o `INFINITY`; una o tutte le lettere possono essere minuscole. NaN può essere rappresentato come una stringa `NAN` (ignorando nuovamente il fatto che le lettere siano maiuscole o minuscole), eventualmente seguita da una coppia di parentesi. Le parentesi possono essere vuote o possono contenere una serie di caratteri, dove ogni carattere è una lettera o un numero.

una cifra o il carattere underscore. Il carattere può essere usato per specificare almeno un bit nella rappresentazione binaria del valore NaN, ma il loro esatto valore è definito dall'implementazione. Lo stesso tipo di sequenza dei caratteri, che lo standard C chiama *n-char-sequence*, viene usata anche nelle chiamate alla funzione `nan` [funzione `nan` > 23.4].

\*D: Lei ha detto che eseguire la chiamata `exit(n)` in qualsiasi punto del programma, solitamente è equivalente a eseguire l'istruzione `return n;` nel main. Quando le due chiamate non sono equivalenti? [p. 713]

R: Ci sono due questioni. La prima: quando la funzione `main` termina, la vita delle sue variabili locali ha termine (assumendo che abbiano una durata della memoria automatica [durata della memorizzazione automatica > 18.2]), ovvero a meno che non vengano dichiarate `static`), ma questo non è vero se viene chiamata la funzione `exit`. Un problema si verifica se una qualsiasi azione che ha luogo al termine del programma (come chiamare una funzione precedentemente registrata usando `atexit` o svuotare uno stream di buffer) richiede l'accesso a una di queste variabili. In particolare, un programma può aver chiamato la `setvbuf` [funzione `setvbuf` > 22.2] e usato una delle variabili del `main` come buffer. Quindi in rari casi un programma potrebbe comportarsi in modo improprio se cerca di terminare dal `main`, mentre funzionerebbe correttamente se invocasse la funzione `exit`.

C99

L'altra questione si verifica nel C99, che rende possibile per il `main` avere un tipo restituito diverso da `int` nel caso l'implementazione permetta specificatamente al compilatore di farlo. In questa circostanza la chiamata `exit(n)` non è necessariamente equivalente a eseguire `return n;` all'interno del `main`. Infatti l'istruzione `return n;` dovrebbe non essere ammessa (se per esempio è stato dichiarato che il `main` restituisce il tipo `void`).

\*D: Vi è qualche relazione tra la funzione `abort` e il segnale SIGABRT? [p. 713]

R: Sì. Agli effetti pratici una chiamata alla funzione `abort` genera il segnale SIGABRT, ma non è presente nessun handler per SIGABRT, il programma termina in modo anomalo come descritto nella Sezione 26.2. Se per SIGABRT è stato installato un handler (chiamato la funzione `signal` [funzione `signal` > 24.3]), questo viene chiamato. Se l'handler non termina, il programma termina in modo anomale. Tuttavia, se l'handler non termina (per esempio chiama la funzione `longjmp` [funzione `longjmp` > 24.4]) allora non termina nemmeno il programma.

D: Perché esistono le funzioni `div` e `ldiv`? Non possiamo usare semplicemente gli operatori / e %? [p. 717]

R: Le funzioni `div` e `ldiv` non sono esattamente uguali agli operatori / e %. Ricordate dalla Sezione 4.1 che nel C89 applicare / e % a operandi negativi non fornisce risultato portatile. Se `i` o `j` sono negativi, dipende dall'implementazione che il valore `i / j` venga arrotondato per eccesso o per difetto. Discorso analogo per il segno di `j`. Il risultato calcolato da `div` e `ldiv`, d'altro canto, non dipende dall'implementazione. Il quoziente viene arrotondato verso lo zero, il resto viene calcolato in accordo con la formula  $n = q \times d + r$ , dove  $n$  è il numero originale,  $q$  è il quoziente,  $d$  è il dividendo ed  $r$  è il resto. Ecco alcuni esempi:

| <i>n</i> | <i>d</i> | <i>q</i> | <i>r</i> |
|----------|----------|----------|----------|
| 7        | 3        | 2        | 1        |
| -7       | 3        | -2       | -1       |
| 7        | -3       | -2       | 1        |
| -7       | -3       | 2        | -1       |

C99

Nel C99 c'è la garanzia che gli operatori / e % producano lo stesso risultato delle funzioni div e ldiv.

L'efficienza è la ragione dell'esistenza delle funzioni div e ldiv. Molte macchine possono avere un'istruzione che calcola sia il quoziente che il resto, quindi chiama-re la funzione div o la ldiv può essere più veloce rispetto a usare separatamente gli operatori / e %.

#### D: Da dove proviene il nome della funzione gmtime? [p. 722]

R: Il nome gmtime sta per *Greenwich Mean Time* (GMT), che si riferisce al tempo locale (solare) al *Royal Observatory* di Greenwich in Inghilterra. Nel 1884 il GMT è stato adottato come riferimento internazionale per l'ora, con altri fusi orari espressi come ore "in ritardo rispetto a GMT" o "in anticipo rispetto a GMT". Nel 1972, il *Coordinated Universal Time* (UTC), ovvero un sistema basato su orologi atomici invece che su osservazioni solari, ha rimpiazzato il GMT come riferimento internazionale. Aggiungendo un "salto" di un secondo ogni pochi anni, l'UTC viene mantenuto sincronizzato con il GMT entro 0.9 secondi. Per tutte le misure di tempo eccetto le più precise, i due sistemi sono da considerare identici.

## Esercizi

### Sezione 26.1

- Riscrivete la funzione `max_int` in modo che, invece di passare il numero di interi come primo argomento, si debba fornire il valore 0 come ultimo argomento.  
*Suggerimento:* la `max_int` deve avere almeno un parametro "normale", quindi non potete rimuovere il parametro `n`. Assumete invece che rappresenti uno dei numeri che deve essere confrontato.
- Scrivete una versione semplificata della `printf` nella quale l'unica specifica di conversione è la `%d` e tutti gli argomenti dopo il primo sono assunti di tipo `int`. Se la funzione incontra un carattere `%` che non è immediatamente seguito da un carattere `d`, deve ignorarli entrambi. La funzione deve utilizzare delle chiamate alla `putchar` per produrre tutti gli output. Potete assumere che la stringa di formato non contenga sequenze di escape.
- Estendete la funzione dell'Esercizio 2 in modo che ammetta le due specifiche di conversione `%d` e `%s`. Ogni `%d` nella stringa di formato indica un argomento `int`, mentre ogni `%s` indica un argomento `char *` (stringa).
- Scrivete una funzione `display` che accetti un qualsiasi numero di argomenti. Il primo argomento deve essere un intero. Gli argomenti rimanenti saranno delle stringhe. Il primo argomento specifica quante stringhe conterrà la chiamata. La funzione stamperà la stringa su una singola riga, con le stringhe adiacenti separate da uno spazio. Per esempio, la chiamata

```
display(4, "Special", "Agent", "Dale", "Cooper");
```

produrrà il seguente output:

Special Agent Dale Cooper

5. Scrivete la seguente funzione:

```
char *vstrcat(const char *first, ...);
```

Tutti gli argomenti della vstrcat si assume siano stringhe, a eccezione dell'ultimo argomento che deve essere un puntatore nullo (con un cast al tipo char \*). La funzione restituisce un puntatore a una stringa allocata dinamicamente contenente la concatenazione degli argomenti. La vstrcat deve restituire un puntatore nullo se non è disponibile sufficiente memoria. Suggerimento: fate in modo che la vstrcat attraversi due volte argomenti: una volta per determinare la quantità di memoria richiesta per la stringa da restituire e una per copiare gli argomenti nella stringa stessa.

6. Scrivete la seguente funzione:

```
char *max_pair(int num_pairs, ...);
```

Gli argomenti della funzione max\_pair si assume siano coppie di interi e stringhe. Il valore di num\_pairs indica quante coppie seguono (una coppia consiste di un argomento int seguito da un argomento char \*). La funzione cerca all'interno delle coppie gli interi per trovare il più grande e poi restituisce l'argomento stringa seguente. Considerate la chiamata seguente:

```
max_pair(5, 180, "Seinfeld", 180, "I Love Lucy",
39, "The Honeymooners", 210, "All in the Family",
86, "The Sopranos")
```

Il più grande argomento int è 210 e quindi la funzione restituirà "All in the Family", che lo segue nell'elenco dell'argomento.

#### Sezione 26.2



7. Spiegate il significato dell'istruzione seguente, assumendo che value sia una variabile di tipo long int e p una variabile di tipo char \*:
- ```
value = strtol(p, &p, 10);
```
8. Scrivete un'istruzione che assegna casualmente alla variabile n uno dei numeri 11, 15 e 19.
9. Scrivete una funzione che restituisca il valore casuale double d nell'intervallo 0 ≤ d < 1.0.
10. Convertite le seguenti chiamate alle funzioni atoi, atol e atoll rispettivamente nelle chiamate alla strtol, strtol e strtoll.
- atoi(str)
 - atol(str)
 - atoll(str)
11. Sebbene la funzione bsearch venga normalmente usata con un vettore ordinato, delle volte funziona correttamente anche con un vettore che è parzialmente ordinato.

dinato. Quale condizione deve soddisfare il vettore per garantire che la `bsearch` funzioni a dovere per una particolare chiave? *Suggerimento:* la risposta compare nello standard C.

- Sezione 26.3**
12. Scrivete una funzione che, quando le viene passato un anno, restituisce un valore `time_t` rappresentante le ore 12:00 a.m. del primo giorno dell'anno.
 13. La Sezione 26.3 descrive alcuni dei formati per la data e l'ora dello standard ISO 8601. Eccone delle altre:
 - (a) gli anni seguiti dal giorno dell'anno: `YYYY-DDD`, dove la `DDD` è un numero compreso tra 001 e 366;
 - (b) anno, settimana e giorno della settimana: `YYYY-Wuuu-D`, dove `uuu` è un numero compreso tra 01 e 53, e `D` è una cifra compresa tra 1 e 7, a partire dal lunedì alla domenica;
 - (c) data e ora combinate: `YYYY-MM-DDThh:mm:ss`.

Fornite alla `strftime` delle stringhe che corrispondano a ognuno di questi formati.

Progetti di programmazione

- W 1. (a) Scrivete un programma che chiami 1000 volte la funzione `rand`, stampando un bit meno significativo di ogni valore restituito (0 se il valore restituito è pari, 1 se è dispari). Individuate dei pattern? (spesso gli ultimi bit del valore restituito dalla funzione `rand` non sono particolarmente casuali).
- W 1. (b) Come potete migliorare la casualità della funzione `rand` per generare dei numeri all'interno di un piccolo intervallo?
- 2. Scrivete un programma che testi la funzione `atexit`. Il programma dovrebbe avere due funzioni (in aggiunta alla funzione `main`): una che stampa `That's all`, e l'altra che stampa `folks!`. Utilizzate la `atexit` per registrare entrambe le funzioni in modo che vengano chiamate al termine del programma. Assicuratevi che queste vengano chiamate nell'ordine corretto, ovvero in modo da poter vedere sullo schermo il messaggio `That's all, folks!`.
- W 3. Scrivete un programma che usi la funzione `clock` per misurare quanto tempo impiega la `qsort` per ordinare un vettore di 1000 interi che sono originariamente in ordine inverso. Eseguite i programmi anche per vettori di 10000 e 100000 interi.
- W 4. Scrivete un programma che chieda all'utente una data (mese, giorno e anno) e un intero `n`. Il programma successivamente dovrà stampare la data di `n` giorni dopo.
- 5. Scrivete un programma che chieda all'utente di immettere due date e poi stampare la differenza tra esse misurata in giorni. *Suggerimento:* usate le funzioni `mktime` e `difftime`.

- W 6. Scrivete dei programmi che visualizzino la data e l'ora correnti in ognuno dei seguenti formati. Utilizzate la funzione `strftime` per effettuare tutta la formattazione o la maggior parte.
- (a) Sunday, June 3, 2007 05:48p
 - (b) Sun, 3 Jun 07 17:48
 - (c) 06/03/07 5:48:34 PM

27 Supporto aggiuntivo del C99 per la matematica

Questo capitolo completa la trattazione della libreria standard descrivendo cinque nuovi header che sono stati introdotti dal C99. Questi header, come alcuni di quelli vecchi, forniscono un supporto per lavorare con i numeri, ma rispetto a questi ultimi sono più specializzati. Alcuni header attraranno principalmente ingegneri, scienziati e matematici che possono aver bisogno dei numeri complessi e di un maggiore controllo sulla rappresentazione dei numeri e sulle modalità di esecuzione dell'aritmetica a virgola mobile.

Le prime due sezioni trattano gli header relativi ai tipi interi. L'header `<stdint.h>` (Sezione 27.1) dichiara dei tipi interi che possiedono uno specificato numero di bit. L'header `<inttypes.h>` (Sezione 27.2) fornisce delle macro che sono utili per leggere e scrivere i valori appartenenti ai tipi definiti in `<stdint.h>`.

Le prossime due sezioni descrivono il supporto del C99 per i numeri complessi. La Sezione 27.3 include una breve spiegazione dei numeri complessi, così come una discussione dei tipi complessi del C99. La Sezione 27.4 tratta l'header `<complex.h>` che fornisce alcune funzioni per eseguire operazioni matematiche sui numeri complessi.

Gli header discussi nelle ultime due sezioni sono relativi ai tipi a virgola mobile. L'header `<tgmath.h>` (Sezione 27.5) fornisce delle macro di tipo generico che facilitano la chiamata alle funzioni di libreria appartenenti a `<complex.h>` e `<math.h>`. Le funzioni nell'header `<fenv.h>` (Sezione 27.6) danno ai programmi l'accesso ai flag di stato e ai modi di controllo.

27.1 L'header `<stdint.h>` (C99): tipi interi

L'header `<stdint.h>` dichiara i tipi interi contenenti uno specificato numero di bit. In aggiunta, questo header definisce delle macro che rappresentano il valore minimo e quello massimo per questi tipi e per quelli dichiarati in altri header (queste macro aumentano quelle presenti nell'header `<limits.h>` [header `<limits.h>` > 23.2]). L'header `<sdtint.h>` definisce anche delle macro parametriche che costituiscono delle costanti intere di uno specifico tipo. In questo header non ci sono funzioni.

La motivazione principale per l'header `<stdint.h>` risiede in un'osservazione fatta nella Sezione 7.5, che discuteva il ruolo delle definizioni dei tipi nel rendere i programmi portabili. Per esempio, se `i` è una variabile `int`, l'assegnamento

```
i = 100000;
```

è corretto se `int` è un tipo a 32 bit mentre non lo è se `int` è un tipo a 16 bit. Il problema è che lo standard C non specifica esattamente quanti bit debba avere un valore `int`. Lo standard *garantisce* che i valori del tipo `int` debbano includere tutti i numeri compresi tra -32767 e $+32767$ (il che richiede almeno 16 bit), ma questo è tutto quello che dice sulla questione. Nel caso della variabile `i`, che deve contenere il numero `100000`, la soluzione tradizionale è quella di dichiarare la variabile di qualche tipo `T`, dove `T` è un nome di tipo creato usando `typedef`. La dichiarazione di `T` può essere adattata in base alle dimensioni degli interi in una particolare implementazione (su una macchina a 16 bit, `T` dovrebbe corrispondere a `long int`, mentre su una macchina a 32 bit può benissimo corrispondere al tipo `int`). Questa è la strategia discussa nella Sezione 7.5.

Se il vostro compilatore supporta il C99, c'è una tecnica migliore. L'header `<stdint.h>` dichiara i nomi per i tipi basati sulla **dimensione** del tipo (il numero di bit utilizzati per salvare i valori del tipo, includendo il bit di segno se presente **[bit di segno > 7.1]**). I nomi `typedef` dichiarati in `<stdint.h>` possono riferirsi ai tipi base (come `int`, `unsigned int` e `long int`) o a dei tipi interi estesi che sono supportati da una particolare implementazione.

Tipi `<stdint.h>`

I tipi dichiarati nell'header `<stdint.h>` ricadono in cinque gruppi.

- **Tipi interi di dimensione esatta.** Ogni nome della forma `intN_t` rappresenta un tipo intero con segno con N bit, memorizzato in complemento a due (il complemento a due è una tecnica usata per rappresentare in binario gli interi con segno, è quasi universale nel mondo dei computer). Per esempio, un valore di tipo `int16_t` sarebbe un intero con segno di 16 bit. Un nome della forma `uintN_t` rappresenta un tipo intero senza segno con N bit. Un'implementazione deve fornire i tipi `intN_t` e `uintN_t` per $N = 8, 16, 32$ e 64 , se supporta interi di queste dimensioni.
- **Tipi interi di dimensione minima.** Ogni nome della forma `int_leastN_t` rappresenta un tipo intero con segno con almeno N bit. Un nome della forma `uint_leastN_t` rappresenta un tipo intero senza segno con N o più bit. L'header `<stdint.h>` deve fornire come minimo i seguenti tipi:

```
int_least8_t    uint_least8_t
int_least16_t   uint_least16_t
int_least32_t   uint_least32_t
int_least64_t   uint_least64_t
```

- **Tipi interi di dimensione minima più veloci.** Ogni nome della forma `int_fastN_t` rappresenta il più veloce tipo intero con almeno N bit. Il significato di "più veloce" dipende dall'implementazione. Se non c'è ragione di classificare un tipo particolare come il più veloce, l'implementazione può scegliere un qualsiasi tipo intero con segno con almeno N bit. Ogni nome della forma `uint_fastN_t` rappresenta il più veloce tipo intero senza segno con N o più bit. L'header `<stdint.h>` deve fornire almeno i seguenti tipi:

```

int_fast8_t    uint_fast8_t
int_fast16_t   uint_fast16_t
int_fast32_t   uint_fast32_t
int_fast64_t   uint_fast64_t

```

- **Tipi interi in grado di contenere oggetti puntatore.** Il tipo `intptr_t` rappresenta un tipo intero con segno che sia in grado di contenere con sicurezza un qualsiasi valore `void *`. Più precisamente, se un puntatore `void *` viene convertito al tipo `intptr_t` e poi riconvertito al tipo `void *`, il puntatore risultante è quello originale, se confrontati, dovranno risultare uguali. Il tipo `uintptr_t` è un tipo intero senza segno con le stesse proprietà di `intptr_t`. L'header `<stdint.h>` non deve necessariamente fornire alcun tipo.
- **Tipi interi con la dimensione più grande.** `intmax_t` è un tipo intero con segno che include tutti i valori che appartengono a un qualsiasi altro tipo con segno. `uintmax_t` è un tipo intero senza segno che include tutti i valori che appartenono a un qualsiasi altro tipo intero senza segno. L'header `<stdint.h>` deve fornire entrambi i tipi, i quali possono essere più grandi del `long long int`.

I nomi dei primi tre gruppi sono dichiarati usando `typedef`.

Un'implementazione può fornire dei tipi interi di dimensione esatta, tipi interi di dimensione minima e tipi interi di dimensione minima più veloci per valori di N aggiunta a quelli sopra elencati. Inoltre non viene richiesto che sia una potenza di 2 (sebbene normalmente sarà un multiplo di 8). Per esempio, un'implementazione può fornire tipi chiamati `int24_t` e `uint24_t`.

Limiti dei tipi interi con dimensione specificata

Per ogni tipo intero con segno dichiarato in `<stdint.h>`, l'header definisce delle macro che specificano i valori minimo e massimo del tipo stesso. Per ogni tipo intero senza segno, l'header `<stdint.h>` definisce una macro che specifica il massimo valore del tipo. Le tre righe della Tabella 27.1 mostrano il valore di queste macro per i tipi interi con dimensione esatta. Le righe rimanenti mostrano i vincoli imposti dallo standard C99 sui valori minimo e massimo per gli altri tipi `<stdint.h>` (i valori precisi di queste macro sono definiti dall'implementazione). Tutte le macro della tabella rappresentano delle espressioni costanti.

Tabella 27.1 Macro per i limiti dei tipi interi con dimensione specificata presenti in `<stdint.h>`

Nome	Valore	Descrizione
<code>INTN_MIN</code>	$-(2^{N-1})$	Minimo valore <code>intN_t</code>
<code>INTN_MAX</code>	$2^{N-1}-1$	Massimo valore <code>intN_t</code>
<code>UINTN_MAX</code>	2^N-1	Massimo valore <code>uintN_t</code>
<code>INT_LEASTN_MIN</code>	$\leq -(2^{N-1}-1)$	Minimo valore <code>int_leastN_t</code>
<code>INT_LEASTN_MAX</code>	$\geq 2^{N-1}-1$	Massimo valore <code>int_leastN_t</code>
<code>UINT_LEASTN_MAX</code>	$\geq 2^N-1$	Massimo valore <code>uint_leastN_t</code>

Nome	Valore	Descrizione
<code>INT_FASTN_MIN</code>	$\leq -(2^{N-1}-1)$	Minimo valore <code>int_fastN_t</code>
<code>INT_FASTN_MAX</code>	$\geq 2^{N-1}-1$	Massimo valore <code>int_fastN_t</code>
<code>UINT_FASTN_MAX</code>	$\geq 2^N-1$	Massimo valore <code>uint_fastN_t</code>
<code>INTPTR_MIN</code>	$\leq -(2^{15}-1)$	Minimo valore <code>intptr_t</code>
<code>INTPTR_MAX</code>	$\geq 2^{15}-1$	Massimo valore <code>intptr_t</code>
<code>UINTPTR_MAX</code>	$\geq 2^{16}-1$	Massimo valore <code>uintptr_t</code>
<code>INTMAX_MIN</code>	$\leq -(2^{63}-1)$	Minimo valore <code>intmax_t</code>
<code>INTMAX_MAX</code>	$\geq 2^{63}-1$	Massimo valore <code>intmax_t</code>
<code>UINTMAX_MAX</code>	$\geq 2^{64}-1$	Massimo valore <code>uintmax_t</code>

Limiti per gli altri tipi interi

Quando il comitato C99 ha creato l'header `<stdint.h>`, ha deciso che questo sarebbe stato un ottimo luogo per mettere le macro che descrivono i limiti dei tipi interi che vanno oltre a quelli dichiarati nello stesso `<stdint.h>`. Questi tipi sono `ptrdiff_t`, `size_t` e `wchar_t` (che appartengono all'header `<stddef.h>` [header `<stddef.h>` > 21.4]), `sig_atomic_t` (dichiarato nell'header `<signal.h>` [header `<signal.h>` > 24.3]), e `wint_t` (dichiarato nell'header `<wchar.h>` [header `<wchar.h>` > 25.5]). La Tabella 27.2 elenca queste macro e mostra il valore di ognuna (o di ogni vincolo sul valore imposto dallo standard C99). In alcuni casi i vincoli sul valore minimo e su quello massimo di un tipo dipendono dal fatto che il tipo stesso sia con o senza segno. Le macro presenti nella Tabella 27.2, come quelle della Tabella 27.1, rappresentano delle espressioni costanti.

Tabella 27.2 Macro per i limiti dei tipi interi con dimensione specificata presenti in `<stdint.h>`

Nome	Valore	Descrizione	
<code>PTRDIFF_MIN</code>	≤ -65535	Minimo valore <code>ptrdiff_t</code>	
<code>PTRDIFF_MAX</code>	$\geq +65535$	Massimo valore <code>ptrdiff_t</code>	
<code>SIG_ATOMIC_MIN</code>	≤ -127 0	(se con segno) (se senza segno)	Minimo valore <code>sig_atomic_t</code>
<code>SIG_ATOMIC_MAX</code>	$\geq +127$ ≥ 255	(se con segno) (se senza segno)	Massimo valore <code>sig_atomic_t</code>
<code>SIZE_MAX</code>	≥ 65535	Massimo valore <code>size_t</code>	
<code>WCHAR_MIN</code>	≤ -127 0	(se con segno) (se senza segno)	Minimo valore <code>wchar_t</code>
<code>WCHAR_MAX</code>	$\geq +127$ ≥ 255	(se con segno) (se senza segno)	Massimo valore <code>wchar_t</code>

Nome	Valore	Descrizione
WINT_MIN	≤ -32767 (se con segno) 0 (se senza segno)	Minimo valore wint_t
WINT_MAX	$\geq +32767$ (se con segno) ≥ 65535 (se senza segno)	Massimo valore wint_t

Macro per le costanti intere

L'header `<stdint.h>` fornisce anche macro parametriche che sono in grado di convertire una costante intera [**costanti intere > 7.1**] (espressa in formato decimale, ottale o esadecimale, ma senza i suffissi U e/o L) in una espressione costante appartenente al tipo intero di dimensione minima o dimensione massima.

Per ogni tipo `int_leastN_t` dichiarato in `<stdint.h>`, l'header definisce una macro parametrica chiamata `INTN_C` che converte una costante intera al tipo stesso (eventualmente usando le promozioni intere [**promozioni intere > 7.4**]). Per ogni tipo `uint_leastN_t` è presente una macro parametrica simile chiamata `UINTN_C`. Queste macro, tra le altre cose, sono utili per inizializzare le variabili. Per esempio, se `i` è una variabile di tipo `int_least32_t`, scrivere

```
i = 100000;
```

è potenzialmente problematico perché la costante 100000 potrebbe essere troppo grande per essere rappresentabile usando il tipo `int` (se `int` fosse un tipo a 16 bit).

Tuttavia, l'istruzione

```
i = INT32_C(100000);
```

è sicura. Se `int_least32_t` rappresenta il tipo `int`, allora `INT32_C(100000)` è di tipo `int`. Se invece `int_least32_t` corrisponde al tipo `long int`, `INT32_C(100000)` è di tipo `long int`.

L'header `<stdint.h>` possiede altre due macro parametriche. La macro `INTMAX_C` converte una costante intera al tipo `intmax_t`, mentre la macro `UINTMAX_C` converte una costante intera al tipo `uintmax_t`.

27.2 L'header `<inttypes.h>` (C99): conversione di formato dei tipi interi

L'header `<inttypes.h>` è strettamente collegato all'header `<stdint.h>`, ovvero l'argomento della Sezione 27.1. Infatti `<inttypes.h>` include `<stdint.h>` e quindi i programmi che includono `<inttypes.h>` non hanno bisogno di includere anche `<stdint.h>`. L'header `<inttypes.h>` estende l'header `<stdint.h>` in due modi. Per prima cosa definisce delle macro che possono essere usate dalle stringhe di formato delle funzioni ...printf e ...scanf per le operazioni di input/output dei tipi interi dichiarati in `<stdint.h>`. Secondariamente, l'header fornisce delle funzioni per lavorare con gli interi della dimensione più grande.

Macro per specificatori di formato

I tipi dichiarati nell'header <stdint.h> possono essere usati per rendere i programmi più portabili, ma possono procurare nuovi mal di testa ai programmatori. Considerate il problema di visualizzare il valore della variabile *i*, dove *i* è di tipo *int_least32_t*. L'istruzione

```
printf("i = %d\n", i);
```

potrebbe non funzionare perché *i* non è necessariamente di tipo *int*. Se *int_least32_t* è un altro nome per il tipo *long int*, allora la specifica di conversione corretta sarebbe *%ld* e non *%d*. Al fine di poter usare le funzioni ...printf e ...scanf in maniera portabile, abbiamo bisogno di un modo per scrivere le specifiche di conversione che corrispondano a ognuno dei tipi dichiarati in <stdint.h>. Ecco dove entra in gioco l'header <inttypes.h>. Per ogni tipo <stdint.h>, l'header <inttypes.h> fornisce una macro che si espande in una stringa letterale contenente lo specificatore di conversione adatto a quel tipo.

Ogni nome di macro è composto da tre parti:

- il nome inizia con PRI o SCN a seconda che la macro venga usata in una chiamata a una funzione ...printf o ...scanf;
- successivamente è presente una lettera che fa da specificatore di conversione (*d* o *i* per i tipi con segno, *o*, *u*, *x* o *X* per i tipi senza segno);
- l'ultima parte del nome indica quale tipo <stdint.h> è coinvolto. Per esempio, il nome di una macro che corrisponde al tipo *int_leastN_t* termina con LEAST*N*.

Ritorniamo al nostro esempio precedente che riguardava la visualizzazione di un intero del tipo *int_least32_t*. Invece di usare la *d* come specificatore di conversione, passeremo all'uso della macro PRIdLEAST32. Per usare questa macro divideremo la stringa di formato della printf in tre pezzi e sostituiremo la *d* presente in *%d* con PRIdLEAST32:

```
printf("i = %" PRIdLEAST32 "\n", i);
```

Probabilmente il valore di PRIdLEAST32 corrisponde a "d" (se il tipo *int_least32_t* è equivalente al tipo *int*) oppure a "ld" (se il tipo *int_least32_t* è equivalente al tipo *long int*). Assumiamo che il valore corrispondente sia "ld". Dopo la sostituzione della macro, l'istruzione diventa:

```
printf("i = %" "ld" "\n", i);
```

Una volta che il compilatore congiunge le tre stringhe letterali per formarne una sola (cosa che farà automaticamente), l'istruzione si presenterà in questo modo:

```
printf("i = %ld\n", i);
```

Osservate che nella nostra specifica di conversione possiamo ancora includere dei flag, un campo di larghezza e le altre opzioni. La macro PRIdLEAST32 fornisce solamente lo specificatore di conversione ed eventualmente anche un modificatore di lunghezza come la lettera *l*.

La Tabella 27.3 elenca le macro presenti in <inttypes.h>.

Tabella 27.3 Macro per gli specificatori di formato presenti in <inttypes.h>

printf Macro per gli interi con segno				
PRIdN	PRIdLEASTN	PRIdFASTN	PRIdMAX	PRIdPTR
PRIiN	PRIiLEASTN	PRIiFASTN	PRIiMAX	PRIiPTR
printf Macro per gli interi senza segno				
PRIoN	PRIoLEASTN	PRIoFASTN	PRIoMAX	PRIoPTR
PRIuN	PRIuLEASTN	PRIuFASTN	PRIuMAX	PRIuPTR
PRIxN	PRIxLEASTN	PRIxFASTN	PRIxMAX	PRIxPTR
PRIXN	PRIXLEASTN	PRIXFASTN	PRIXMAX	PRIXPTR
scanf Macro per gli interi con segno				
SCNdN	SCNdLEASTN	SCNdFASTN	SCNdMAX	SCNdPTR
SCNiN	SCNiLEASTN	SCNiFASTN	SCNiMAX	SCNiPTR
scanf Macro per gli interi senza segno				
SCNoN	SCNoLEASTN	SCNoFASTN	SCNoMAX	SCNoPTR
SCNuN	SCNuLEASTN	SCNuFASTN	SCNuMAX	SCNuPTR
SCNxN	SCNxLEASTN	SCNxFASTN	SCNxMAX	SCNxPTR

Funzioni per i tipi interi con la dimensione più grande

```
intmax_t imaxabs(intmax_t j);
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
intmax_t strtoimax(const char * restrict nptx,
                    char ** restrict endptr,
                    int base);
uintmax_t strtouimax(const char * restrict nptx,
                     char ** restrict endptr,
                     int base);
intmax_t wcstoimax(const wchar_t * restrict nptr,
                    wchar_t ** restrict endptr,
                    int base);
uintmax_t wcstouimax(const wchar_t * restrict nptr,
                     wchar_t ** restrict endptr,
                     int base);
```

Oltre a definire delle macro, l'header <inttypes.h> fornisce anche delle funzioni per lavorare con gli interi con la dimensione più grande. Questi interi sono stati introdotti nella Sezione 27.1. Un intero con la dimensione più grande è di tipo `intmax_t` (il tipo intero con segno avente la dimensione più grande supportata da un'implementazione) o `uintmax_t` (il più grande tipo intero senza segno). Questi tipi possono avere la stessa dimensione del tipo `long long int`, ma possono essere ancora più grandi. Per

esempio, si può avere che il tipo `long long int` sia di 64 bit mentre i tipi `intmax_t` e `uintmax_t` di 128 bit.

`imaxabs`
`imaxdiv`

Le funzioni `imaxabs` e `imaxdiv` sono le versioni per gli interi più grandi delle funzioni per l'aritmetica intera dichiarate nell'header `<stdlib.h>` [header <stdlib.h> 26.2]. La funzione `imaxabs` restituisce il valore assoluto del suo argomento. Sia l'argomento che il valore restituito sono di tipo `intmax_t`. La funzione `imaxdiv` divide il suo primo argomento per il secondo restituendo un valore `imaxdiv_t`. Il tipo `imaxdiv_t` è una struttura che contiene sia un membro quoziante (chiamato `quot`) che un membro resto (chiamato `rem`). Entrambi i membri sono di tipo `intmax_t`.

`strtoimax`
`strtoumax`

Le funzioni `strtoimax` e `strtoumax` sono le versioni per gli interi più grandi delle funzioni di conversione numerica dell'header `<stdlib.h>`. La funzione `strtoimax` è equivalente alle funzioni `strtol` e `strtoll`, ma a differenza di queste restituisce un valore del tipo `uintmax_t`. La funzione `strtoumax` è equivalente alle funzioni `strtoul` e `strtoull`, ma a eccezione di queste restituisce un valore di tipo `uintmax_t`. Sia la `strtoimax` che la `strtoumax` restituiscono il valore zero se non può essere effettuata nessuna conversione. Entrambe le funzioni salvano il valore ERANGE nella variabile `errno` nel caso in cui la conversione producesse un valore al di fuori dell'intervallo del tipo restituito. In aggiunta la funzione `strtoimax` restituisce il più piccolo o il più grande valore `intmax_t` (`INTMAX_MIN` o `INTMAX_MAX`). La funzione `strtoumax` restituisce il più grande valore `uintmax_t`, ovvero `UINTMAX_MAX`.

`wstoiimax`
`wstoumax`

Le funzioni `wcstoimax` e `wcstoumax` sono la versione intera delle funzioni per la conversione numerica delle stringhe wide presenti nell'header `<wchar.h>` [header <wchar.h> 25.5]. La funzione `wcstoimax` è equivalente alle funzioni `wcstol` e `wcstoll`, ma a differenza di queste restituisce un valore del tipo `intmax_t`. La funzione `wcstoumax` è equivalente alle funzioni `wcstoul` e `wcstoull`, ma a differenza di queste restituisce un valore di tipo `uintmax_t`. Sia la `wcstoimax` che la `wcstoumax` restituiscono il valore zero se non può essere effettuata nessuna conversione. Entrambe le funzioni salvano il valore ERANGE nella variabile `errno` nel caso in cui la conversione producesse un valore che si trova al di fuori dell'intervallo del tipo restituito. Inoltre la funzione `wcstoimax` restituisce il più piccolo o il più grande valore `intmax_t` (`INTMAX_MIN` o `INTMAX_MAX`). La `wcstoumax` invece restituisce il più grande valore `uintmax_t`, ovvero `UINTMAX_MAX`.

27.3 Numeri complessi (C99)

I numeri complessi vengono utilizzati nelle applicazioni scientifiche, ingegneristiche e matematiche. Il C99 fornisce diversi tipi complessi, permette agli operatori di trattare degli operandi complessi e aggiunge alla libreria standard un header chiamato `<complex.h>`. Vi è un problema, però: i numeri complessi non sono supportati da tutte le implementazioni del C99. La Sezione 14.3 ha trattato le differenze esistenti tra le implementazioni *hosted* del C99 e le implementazioni *freestanding*. Un'implementazione *hosted* deve accettare tutti i programmi conformi allo standard C99, mentre un'implementazione *freestanding* non deve necessariamente compilare i programmi che usano i tipi complessi o gli header standard oltre a `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>` e `<stdint.h>`. Di conseguenza una implementazione *freestanding* può essere priva sia dei tipi complessi che dell'header `<complex.h>`.

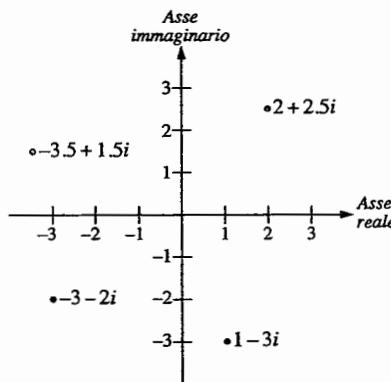
Inizieremo con una rassegna della definizione matematica dei numeri complessi e dell'aritmetica complessa. Successivamente guarderemo ai tipi complessi del C99 e alle operazioni che possono essere effettuate sui valori appartenenti a questi tipi. La trattazione dei numeri complessi continua nella Sezione 27.4, la quale descrive l'header <complex.h>.

Definizione dei numeri complessi

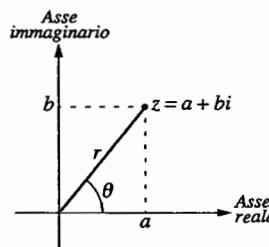
Sia i la radice quadrata di -1 (un numero tale che $i^2 = -1$). Il numero i è conosciuto come l'**unità immaginaria**. Spesso gli ingegneri la rappresentano con il simbolo j invece che i . Un **numero complesso** ha la forma $a + bi$, dove a è detta **parte reale** del numero, mentre b è la **parte immaginaria**. Notate che i numeri complessi includono i numeri reali come un caso particolare (dove $b = 0$).

Perché sono utili i numeri complessi? Per una ragione: permettono soluzioni a problemi che altrimenti sarebbero irrisolvibili. Considerate l'equazione $x^2 + 1 = 0$, la quale non ha soluzione se la x viene ristretta ai numeri reali. Se sono ammessi i numeri complessi ci sono due soluzioni: $x = i$ e $x = -i$.

I numeri complessi possono essere pensati come dei punti in uno spazio bidimensionale conosciuto come **piano complesso**. Ogni numero complesso (un punto nel piano complesso) è rappresentato da coordinate cartesiane, dove la parte reale del numero corrisponde alla coordinata x del punto e la parte immaginaria corrisponde alla coordinata y . Per esempio, i numeri complessi $2 + 2.5i$, $1 - 3i$, $-3 - 2i$ e $-3.5 + 1.5i$ possono essere rappresentati in questo modo:



Anche un sistema alternativo, conosciuto come **coordinate polari**, può essere usato per specificare un punto sul piano complesso. Con le coordinate polari, un numero complesso z viene rappresentato dai valori r e θ , dove r è la lunghezza del segmento che va dall'origine a z , e θ è l'angolo compreso tra questo segmento e l'asse reale:



Il valore r viene chiamato **valore assoluto** di z (è conosciuto anche come *norma*, *modulo* o *magnitudine*), mentre θ viene detto **argomento** (o *angolo di fase*) di z . Il valore assoluto di $a + bi$ è dato dalla seguente equazione:

$$|a + bi| = \sqrt{a^2 + b^2}$$

Per delle informazioni aggiuntive sulla conversione tra le coordinate cartesiane e quelle polari o viceversa, leggete i Progetti di Programmazione presenti alla fine di questo capitolo.

Aritmetica complessa

La somma di due numeri complessi viene calcolata sommando separatamente le parti reali e quelle immaginarie dei due numeri. Per esempio:

$$(3 - 2i) + (1.5 + 3i) = (3 + 1.5) + (-2 + 3)i = 4.5 + i$$

La differenza tra due numeri complessi viene calcolata in maniera simile, ovvero sottraendo separatamente le parti reali e quelle immaginarie. Per esempio:

$$(3 - 2i) - (1.5 + 3i) = (3 - 1.5) + (-2 - 3)i = 1.5 - 5i$$

La moltiplicazione di due numeri complessi viene calcolata moltiplicando ogni termine del primo numero per ogni termine del secondo numero e sommando i prodotti:

$$\begin{aligned} (3 - 2i) \times (1.5 + 3i) &= (3 \times 1.5) + (3 \times 3i) + (-2i \times 1.5) + (-2i \times 3i) \\ &= 4.5 + 9i - 3i - 6i^2 = 10.5 + 6i \end{aligned}$$

Osservate che l'identità $i^2 = -1$ viene usata per semplificare il risultato.

Dividere dei numeri complessi è leggermente più complicato. Prima abbiamo bisogno del concetto di **complesso coniugato** di un numero, che viene trovato cambiando il segno della parte immaginaria del numero stesso. Per esempio: il numero $7 - 4i$ è il coniugato di $7 + 4i$ e $7 + 4i$ è il coniugato di $7 - 4i$. Scriviamo z^* per denotare il complesso coniugato del numero z .

Il quoziente di due numeri complessi y e z è dato dalla formula

$$y/z = yz^*/zz^*$$

Si trova che zz^* è sempre un numero reale, quindi dividere zz^* per yz^* è semplice (semplicemente dividendo separatamente sia la parte reale che quella immaginaria di yz^*). L'esempio seguente mostra come dividere il numero $10.5 + 6i$ per $3 - 2i$:

$$\frac{10.5 + 6i}{3 - 2i} = \frac{(10.5 + 6i)(3 + 2i)}{(3 - 2i)(3 + 2i)} = \frac{19.5 + 39i}{13} = 1.5 + 3i$$

Tipi complessi nel C99

Lo standard C99 incorpora un supporto considerevole per i numeri complessi. Senza includere nessun header di libreria possiamo dichiarare delle variabili che rappresentano dei numeri complessi ed eseguire dell'aritmetica e altre operazioni su di esse.

Il C99 fornisce tre tipi complessi che sono stati introdotti per la prima volta nella Sezione 7.2: `float _Complex`, `double _Complex` e `long double _Complex`. Questi possono essere usati come gli altri tipi del C, ovvero per dichiarare variabili, parametri, tipi restituiti, elementi di vettori, membri di strutture e unioni, e così via. Per esempio, possiamo dichiarare tre variabili nel modo seguente:

```
float _Complex x;  
double _Complex y;  
long double _Complex z;
```

Ognuna di queste variabili viene memorizzata semplicemente come un vettore di due normali numeri a virgola mobile. Quindi, `y` viene memorizzata come due valori `double` adiacenti, con il primo di questi contenente la parte reale di `y` e la seconda contenente la parte immaginaria.

Il C99 permette alle implementazioni di fornire anche i tipi immaginari (la keyword `_Imaginary` è riservata a questo scopo) ma non ne fa un obbligo.

Operazioni sui numeri complessi

I numeri complessi possono essere usati nelle espressioni, sebbene solo gli operatori seguenti ammettono degli operandi complessi:

- + e - unari
- Negazione logica (!)
- sizeof
- Cast
- Moltiplicativi (* e /)
- Additivi (+ e -)
- Uguaglianza (== e !=)
- and logico (&&)
- or logico (||)
- Condizionale (?:)
- Assegnamento semplice (=)
- Assegnamento composto (solo *=, /=, += e -=)
- Virgola (,)

Le assenze più evidenti nell'elenco includono gli operatori relazionali (<, <=, > e >=), assieme con gli operatori di incremento (++) e decremento (--).

Regole di conversione per i tipi complessi

La Sezione 7.4 ha parlato delle regole del C99 per le conversioni di tipo, ma senza trattare i tipi complessi. Ora è venuto il momento di porre riparo a questa situazione. Prima di addentrarci nelle regole di conversione, però, abbiamo bisogno di una nuova terminologia. Per ogni tipo a virgola mobile c'è un **tipo reale corrispondente**. Nel caso dei tipi reali a virgola mobile (float, double e long double), il tipo reale corrispondente è lo stesso del tipo originale. Per i tipi complessi, il tipo reale corrispondente è il tipo originale senza la parola _Complex (per esempio il tipo reale corrispondente a float _Complex è float).

Ora siamo pronti per discutere delle regole generali che governano le conversioni di tipo che coinvolgono i tipi complessi. Raggrupperemo queste regole in tre categorie.

- **Da complesso a complesso.** La prima regola concerne le conversioni da un tipo complesso a un altro, come la conversione dal tipo float _Complex al tipo double _Complex. In questa situazione, le parti reali e immaginarie vengono convertite separatamente usando le regole per i tipi reali corrispondenti (leggete la Sezione 7.4). Nel nostro esempio, la parte reale del valore float _Complex viene convertita al tipo double, ottenendo così la parte reale del valore double _Complex. La parte immaginaria viene convertita al tipo double in maniera simile.
- **Da reale a complesso.** Quando un valore di un tipo reale viene convertito in un tipo complesso, la parte reale del numero viene convertita usando le regole per la conversione da un tipo reale a un altro. La parte immaginaria del risultato viene imposta allo zero positivo o senza segno.
- **Da complesso a reale.** Quando un valore complesso viene convertito in un tipo reale, la parte immaginaria del numero viene scartata. La parte reale viene convertita usando le regole per la conversione da un tipo reale a un altro.

Un particolare insieme di conversioni di tipo, conosciuto come le normali conversioni aritmetiche, viene automaticamente applicato agli operandi della maggior parte degli operatori binari. Ci sono delle regole speciali per eseguire le normali conversioni aritmetiche quando l'ultimo dei due operandi è di un tipo complesso:

1. se il corrispondente tipo reale di uno degli operandi è long double, viene convertito l'altro operando in modo che il suo tipo reale corrispondente sia long double;
2. altrimenti, se il tipo reale corrispondente di uno degli operandi è double, viene convertito l'altro operando in modo che il suo tipo reale corrispondente sia double;
3. negli altri casi, uno degli operandi deve essere di tipo float così come il suo tipo reale corrispondente. Viene convertito l'altro operando in modo che anche il suo tipo reale corrispondente sia float.

Dopo la conversione, un operando reale appartiene ancora a un tipo reale e un operando complesso appartiene ancora a un tipo complesso.

Normalmente l'obiettivo delle normali conversioni aritmetiche è quello di convertire entrambi gli operandi a un tipo comune. Tuttavia, quando un operando reale

viene mischiato con un operando complesso, eseguire le normali conversioni aritmetiche fa sì che gli operandi possiedano un comune tipo reale, ma non necessariamente lo stesso tipo. Per esempio, sommando un operando float con un operando double _Complex fa sì che l'operando float venga convertito al tipo double invece che al tipo double _Complex. Il tipo del risultato sarà il tipo complesso il cui tipo reale corrispondente combacia con il tipo reale comune. Nel nostro esempio, il tipo del risultato è double _Complex.

27.4 L'header <complex.h> (C99): aritmetica complessa

Come abbiamo visto nella Sezione 27.3, il C99 possiede un supporto significativo per i numeri complessi. L'header <complex.h> fornisce un supporto aggiuntivo sotto forma di funzioni matematiche sui numeri complessi, oltre che con alcune macro molto utili e una direttiva pragma. Per prima cosa concentriamoci sulle macro.

Macro <complex.h>

L'header <complex.h> definisce le macro illustrate nella Tabella 27.4.

Tabella 27.4 Macro presenti in <complex.h>

Nome	Valore
complex	_Complex
_Complex_I	Unità immaginaria, è di tipo const float_Complex
I	_Complex_I

La macro complex funge da nome alternativo per la scomoda keyword _Complex. Abbiamo già visto una situazione come questa con il tipo booleano: il comitato C99 ha scelto la nuova keyword (_Bool) in modo che non danneggi i programmi esistenti, ma ha fornito un nome migliore (bool) sotto forma di una macro definita nell'header <stdbool.h> [header <stdbool.h> 21.5]. I programmi che includono l'header <complex.h> possono usare la parola complex al posto di _Complex, così come i programmi che includono <stdbool.h> possono utilizzare bool invece di _Bool.

La macro I gioca un ruolo importante nel C99. Non c'è nessuna particolare funzionalità per creare un numero complesso a partire dalla sua parte reale e da quella immaginaria. Un numero complesso può essere costruito, invece, moltiplicando la parte immaginaria per I e sommando la parte reale:

```
double complex dc = 2.0 + 3.5 * I;
```

il valore della variabile dc è pari a $2 + 3.5i$.

Osservate che sia la macro _Complex_I che I rappresentano l'unità immaginaria i . Presumibilmente la maggior parte dei programmati utilizzerà I invece che _Complex_I. Tuttavia, visto che I può essere già utilizzata per altri scopi da del codice

preesistente, la macro `_Complex_I` è disponibile come riserva. Se il nome `I` causa conflitti, la sua definizione può sempre essere annullata:

```
#include <complex.h>
#undef I
```

Il programmatore può allora definire un diverso (ma sempre breve) nome per `i`, come

```
#define J _Complex_I
```

Osservate inoltre che il tipo `_Complex_I` (e quindi il tipo di `I`) è `float_Complex` o `double_Complex`. Quando viene usata nelle espressioni, la macro `I` viene automaticamente ingrandita al tipo `double_Complex` o `long double_Complex` se necessario.

La direttiva pragma CX_LIMITED_RANGE

L'header `<complex.h>` fornisce una direttiva pragma [[direttiva pragma > 14.5](#)] chiamata `CX_LIMITED_RANGE` che permette al compilatore di usare le seguenti formule standardizzate per la moltiplicazione, la divisione e il valore assoluto:

$$(a + bi) \times (c + di) = (ac - bd) + (bc + ad)i$$

$$(a + bi) \times (c + di) = [(ac + bd) + (bc - ad)i] \times (c^2 + d^2)$$

$$|a + bi| = \sqrt{a^2 + b^2}$$

In alcuni casi usare queste formule può provocare dei risultati anomali a causa di verificarsi di un overflow o di un underflow. Inoltre queste formule non gestiscono in modo appropriato i valori infiniti. A causa di questi problemi, il C99 non le utilizza senza il permesso del programmatore.

La direttiva `CX_LIMITED_RANGE` presenta il seguente aspetto:

```
#pragma STDC CX_LIMITED_RANGE on-off-switch
```

Dove `on-off-switch` corrisponde a `ON`, `OFF` oppure a `DEFAULT`. Se la direttiva viene usata con il valore `ON`, permette al compilatore di usare le formule sopra elencate. Il valore `OFF` fa sì che il compilatore esegua i calcoli in un modo più sicuro ma che può essere più lento. L'impostazione di default indicata dal valore `DEFAULT` è equivalente a `ON`.

La durata della pragma `CX_LIMITED_RANGE` dipende da dove questa viene usata all'interno del programma. Quando si trova al livello più alto del file sorgente, al di fuori di qualsiasi dichiarazione esterna, rimane attiva fino alla prossima direttiva `CX_LIMITED_RANGE` o alla fine del file. L'unico altro punto dove può comparire questa direttiva è l'inizio di un'istruzione composta (eventualmente il corpo di una funzione). In tal caso la direttiva rimane attiva fino alla prossima pragma `CX_LIMITED_RANGE` (anche in un'istruzione composta annidata) o alla fine dell'istruzione composta. Alla fine dell'istruzione composta, lo stato della direttiva ritorna al valore che possedeva precedentemente all'inizio dell'istruzione stessa.

Funzioni `<complex.h>`

L'header `<complex.h>` fornisce delle funzioni simili a quelle della versione C99 di `<math.h>`. Le funzioni `<complex.h>` sono divise in gruppi proprio come lo sono que-

di `<math.h>`: trigonometriche, iperboliche, esponenziali e logaritmiche, di potenza valore assoluto. Le sole funzioni che sono uniche per i numeri complessi sono le funzioni di manipolazione, l'ultimo gruppo discusso in questa sezione.

Ogni funzione `<complex.h>` è presente in tre versioni: una versione `float complex`, una versione `double complex` e una versione `long double complex`. Il nome delle versioni `float complex` finisce con la lettera `f`, mentre il nome delle versioni `long double complex` termina con la lettere `l`.

Prima di addentrarci nelle funzioni `<complex.h>` facciamo alcuni commenti generali. Per prima cosa, come con le funzioni `<math.h>`, anche le funzioni `<complex.h>` aspettano che gli angoli siano espressi in radianti e non in gradi. Secondariamente, quando si verifica un errore, le funzioni `<complex.h>` possono memorizzare un valore nella variabile `errno` [variabile `errno` > 24.2], ma non sono obbligate a farlo.

C'è un'ultima cosa prima di trattare le funzioni `<complex.h>`. Il termine **punto di diramazione** (*branch cut*) spesso compare nelle descrizioni delle funzioni e possono plausibilmente possedere più di un possibile valore restituito. Nel reale dei numeri complessi, scegliere quale valore restituire crea un punto di diramazione: una curva (spesso semplicemente una linea) sul piano complesso attorno alla quale la funzione è discontinua. Solitamente i punti di diramazione non sono unici, e spesso sono determinati per convenzione. Una definizione esatta di punto di diramazione ci spinge nell'analisi complessa più di quanto si vuole fare in questo libro. Di conseguenza verranno riprodotte le restrizioni dallo standard C99 senza ulteriori spiegazioni.

Funzioni trigonometriche

```
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);

double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);

double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);

double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);

double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);

double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

- cacos** La funzione `cacos` calcola l'arcocoseno complesso, con un punto di diramazione al di fuori dell'intervallo $[-1, +1]$ lungo l'asse reale. Il valore restituito risiede su una striscia priva di confini lungo l'asse immaginario e delimitata dall'intervallo $[0, \pi]$ lungo l'asse reale.
- casin** La funzione `casin` calcola l'arcoseno complesso, con punti di diramazione al di fuori dell'intervallo $[-1, +1]$ lungo l'asse reale. Il valore restituito risiede su una striscia priva di confini sull'asse immaginario e delimitata dall'intervallo $[-\pi/2, +\pi/2]$ lungo l'asse reale.
- catan** La funzione `catan` calcola l'arcotangente complessa, con punti di diramazione al di fuori dell'intervallo $[-i, +i]$ lungo l'asse immaginario. Il valore restituito risiede su una striscia priva di confini sull'asse immaginario e delimitata dall'intervallo $[-\pi/2, +\pi/2]$ lungo l'asse reale.
- ccos**
cas
ctan La funzione `ccos` calcola il coseno complesso, la funzione `csin` calcola il seno complesso e la `ctan` calcola la tangente complessa.

Funzioni iperboliche

```
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);

double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);

double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);

double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);

double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

- cacosh** La funzione `cacosh` calcola l'arcocoseno iperbolico complesso, con un punto di ramificazione nei valori minori di 1 lungo l'intervallo dell'asse reale. Il valore restituito risiede su una mezza striscia di valori non negativi lungo l'asse reale e delimitata dall'intervallo $[-i\pi, +i\pi]$ lungo l'asse immaginario.
- casinh** La funzione `casinh` calcola l'arcoseno iperbolico complesso, con dei punti di ramificazione al di fuori dell'intervallo $[-i, +i]$ lungo l'asse immaginario. Il valore restituito risiede in una striscia priva di confini lungo l'asse reale e delimitata dall'intervallo $[-i\pi/2, +i\pi/2]$ lungo l'asse immaginario.

catanh	La funzione catanh calcola l'arcotangente iperbolica complessa, con dei punti di ramificazione al di fuori dell'intervallo $[-1, +1]$ lungo l'asse reale. Il valore restituito risiede su una striscia priva di confini lungo l'asse reale e nell'intervallo $[-i\pi/2, +i\pi/2]$ lungo l'asse immaginario.
ccosh csinh ctanh	La funzione ccosh calcola il coseno iperbolico complesso, la funzione csinh calcola il seno iperbolico complesso e la funzione ctanh calcola la tangente iperbolica complessa.

Funzioni esponenziali e logaritmiche

```
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);

double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

cexp	La funzione cexp calcola l'esponenziale complesso con base e .
clog	La funzione clog calcola il logaritmo naturale complesso (base e), con un punto di ramificazione lungo la parte negativa dell'asse reale. Il valore restituito risiede su una striscia priva di confini lungo l'asse reale e delimitata dall'intervallo $[-i\pi, +i\pi]$ lungo l'asse immaginario.

Funzioni per le potenze e il valore assoluto

```
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);

double complex cpow(double complex x,
                     double complex y);
float complex cpowf(float complex x,
                     float complex y);
long double complex cpowl(long double complex x,
                           long double complex y);

double complex csqrt(double complex z);
float complex csqrif(float complex z);
long double complex csqril(long double complex z);
```

cabs	La funzione cabs calcola il valore assoluto complesso.
cpow	La funzione cpow restituisce il valore di x elevato alla potenza y . La funzione presenta un punto di ramificazione per il primo parametro lungo la parte negativa dell'asse reale.
csqrt	La funzione csqrt calcola la radice quadrata complessa con un punto di ramificazione lungo la parte negativa dell'asse reale. Il valore restituito dalla funzione risiede nel semipiano destro (includendo l'asse immaginario).

Funzioni di manipolazione

```

double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);

double cimag(double complex z);
float cimaf(float complex z);
long double cimatl(long double complex z);

double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);

double cproj(double complex z);
float cprojf(float complex z);
long double cprojl(long double complex z);

double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);

```

- carg La funzione carg restituisce l'argomento (l'angolo di fase) di z. La funzione presenta un punto di ramificazione lungo la parte negativa dell'asse reale. Il valore restituito risiede nell'intervallo $[-\pi, +\pi]$.
- cimag La funzione cimag restituisce la parte immaginaria di z.
- conj La funzione conj restituisce il complesso coniugato di z.
- cproj La funzione cproj calcola la proiezione di z sulla sfera di Riemann. Il valore restituito è uguale a z, a meno che una delle sue componenti non sia infinita. In tal caso la funzione restituisce INFINITY + I * copysign(0.0, cimag(z)).
- creal La funzione creal restituisce la parte reale di z.

PROGRAMMA

Trovare le radici di un'equazione quadratica

Le radici dell'equazione quadratica

$$ax^2 + bx + c = 0$$

sono date dalla formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In generale il valore di x sarà un numero complesso perché la radice quadrata di $-4ac$ è immaginaria se $b^2 - 4ac$ (anche conosciuto come **discriminante**) è minore di 0.

Per esempio, supponete che $a = 5$, $b = 2$ e $c = 1$, il che ci porta all'equazione

$$5x^2 + 2x + 1 = 0$$

Il valore del discriminante è $4 - 20 = -16$, di conseguenza le radici dell'equazione saranno dei numeri complessi. Il programma seguente, che utilizza diverse funzioni dell'header <complex.h>, calcola e visualizza le radici dell'equazione.

```
quadratic.c /* trova le radici dell'equazione 5x**2 + 2x + 1 = 0 */

#include <complex.h>
#include <stdio.h>

int main(void)
{
    double a = 5, b = 2, c = 1;
    double complex discriminant_sqrt = csqrt(b * b - 4 * a * c);
    double complex root1 = (-b + discriminant_sqrt) / (2 * a);
    double complex root2 = (-b - discriminant_sqrt) / (2 * a);

    printf("root1 = %g + %gi\n", creal(root1), cimag(root1));
    printf("root2 = %g + %gi\n", creal(root2), cimag(root2));

    return 0;
}
```

L'output del programma sarà il seguente:

```
root1 = -0.2 + 0.4i
root2 = -0.2 + -0.4i
```

Il programma quadratic.c mostra come visualizzare un numero complesso estraendo la sua parte reale e quella immaginaria e poi scrivendo ognuna di queste come un numero a virgola mobile. La funzione printf è priva di specificatori di conversione per i numeri complessi e quindi questa è la tecnica più semplice. Inoltre non ci sono scorciatoie per la lettura dei numeri complessi: un programma avrà bisogno di ottenere separatamente le parti reali e immaginarie e poi combinarle in un singolo numero complesso.

27.5 L'header <tgmath.h> (C99): matematica per tipi generici

L'header <tgmath.h> fornisce delle macro parametriche con dei nomi che corrispondono a quelli delle funzioni presenti in <math.h> e <complex.h>. Queste **macro per tipi generici** possono individuare il tipo degli argomenti che vengono loro passati e sostituirsi con una chiamata alla versione appropriata delle funzioni <math.h> e <complex.h>.

Come abbiamo visto nelle sezioni 23.3, 23.4 e 27.4, nel C99 ci sono versioni multiple per molte funzioni matematiche. Per esempio, la funzione sqrt è presente in una versione double (sqrt), in una versione float (sqrtf) e in una versione long double (sqrtl), così come in tre versioni per i numeri complessi (csqrt, csqrft e csqrtl). Usando l'header <tgmath.h>, il programmatore può semplicemente invocare sqrt senza doversi preoccupare di quale versione sia necessaria: sqrt(x) può essere una chiamata a una qualsiasi delle sei versioni della sqrt, a seconda del tipo di x.

Un vantaggio nell'usare <tgmath.h> è che le chiamate alle funzioni matematiche diventano semplici da scrivere (e da leggere!). Cosa più importante è che una chiamata

a una macro per un tipo generico non deve essere modificata se in futuro dovessimo cambiare il tipo del suo argomento.

Tra l'altro l'header <tgmath.h> include sia <math.h> che <complex.h> e quindi includerlo fornirà automaticamente accesso alle funzioni di questi ultimi.

Macro per tipi generici

Le macro per tipi generici definite nell'header <tgmath.h> ricadono in tre gruppi a seconda che queste corrispondano alle funzioni presenti in <math.h>, in <complex.h> o in entrambi gli header.

La Tabella 27.5 elenca le macro per i tipi generici che corrispondono alle funzioni presenti sia in <math.h> che in <complex.h>. Osservate che il nome di ogni macro corrisponde al nome della funzione <math.h> privo di suffisso (per esempio `acos` al posto di `acosf` o `acosl`).

Tabella 27.5 Macro per tipi generici presenti in <tgmath.h> (Gruppo 1)

Funzione <math.h>	Funzione <complex.h>	Macro per tipi generici
<code>acos</code>	<code>cacos</code>	<code>acos</code>
<code>asin</code>	<code>casin</code>	<code>asin</code>
<code>atan</code>	<code>catan</code>	<code>atan</code>
<code>acosh</code>	<code>cacosh</code>	<code>acosh</code>
<code>asinh</code>	<code>casinh</code>	<code>asinh</code>
<code>atanh</code>	<code>catanh</code>	<code>atanh</code>
<code>cos</code>	<code>ccos</code>	<code>cos</code>
<code>sin</code>	<code>csin</code>	<code>sin</code>
<code>tan</code>	<code>ctan</code>	<code>tan</code>
<code>cosh</code>	<code>ccosh</code>	<code>cosh</code>
<code>sinh</code>	<code>csinh</code>	<code>sinh</code>
<code>tanh</code>	<code>ctanh</code>	<code>tanh</code>
<code>exp</code>	<code>cexp</code>	<code>exp</code>
<code>log</code>	<code>clog</code>	<code>log</code>
<code>pow</code>	<code>cpow</code>	<code>pow</code>
<code>sqr</code>	<code>csqr</code>	<code>sqr</code>
<code>fabs</code>	<code>cabs</code>	<code>fabs</code>

Le macro del secondo gruppo (Tabella 27.6) corrispondono solo alle funzioni presenti in <math.h>. Ogni macro possiede lo stesso nome della funzione <math.h> priva di suffisso. Passare un argomento complesso a una di queste macro provoca un comportamento indefinito.

Tabella 27.6 Macro per tipi generici presenti in <tgmath.h> (Gruppo 2)

<code>atan2</code>	<code>fma</code>	<code>llround</code>	<code>remainder</code>
<code>cbrt</code>	<code>fmax</code>	<code>log10</code>	<code>remquo</code>
<code>ceil</code>	<code>fmin</code>	<code>log1p</code>	<code>rint</code>

copysign	fmod	log2	round
erf	frexp	logb	scalbn
erfc	hypot	lrint	scalbln
exp2	ilogb	lround	tgamma
expm1	ldexp	nearbyint	trunc
fdim	lgamma	nextafter	
floor	llrint	nexttoward	

Le macro appartenenti all'ultimo gruppo (Tabella 27.7) corrispondono a funzioni presenti solo nell'header <complex.h>.

Tabella 27.7 Macro per tipi generici presenti in <tgmaht.h> (Gruppo 3)

carg	conj	creal
cimag	cproj	

Con le tre tabelle vengono prese in considerazione tutte le funzioni presenti in <math.h> e <complex.h> che possiedono delle versioni multiple a eccezione della funzione modf.



Invocare una macro per tipi generici

Per capire cosa accade quando viene invocata una macro per tipi generici, abbiamo prima bisogno del concetto di **parametro generico**. Considerate i prototipi per le tre versioni della funzione nextafter (dall'header <math.h>):

```
double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
```

Il tipo di *x* e *y* dipende dalla versione della funzione e quindi entrambi i parametri sono generici. Ora considerate i prototipi per le tre versioni della funzione nexttoward:

```
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

Il primo parametro è generico mentre il secondo non lo è (è sempre di tipo long double). Nella versione senza suffisso della funzione, i parametri generici sono sempre di tipo double (o double complex).

Quando una macro per tipi generici viene invocata, il primo passo è quello di determinare se deve essere rimpiazzata da una funzione <math.h> o da una funzione <complex.h> (questo non si applica alle macro della Tabella 27.6 che vengono sempre sostituite da una funzione <math.h>, o alle macro della Tabella 27.7 che vengono sempre sostituite da una funzione <complex.h>).

La regola è semplice: se uno degli argomenti corrispondenti a un parametro generico è complesso, allora verrà scelta una funzione <complex.h>, altrimenti verrà scelta una funzione <math.h>.

Il passo successivo è quello di dedurre quale versione della funzione `<math.h>` o della funzione `<complex.h>` debba essere chiamata. Assumiamo che la funzione da chiamare appartenga a `<math.h>` (le regole per il caso `<complex.h>` sono analoghe). Vengono applicate le regole seguenti, nell'ordine in cui sono elencate:

1. se uno degli argomenti corrispondenti a un parametro generico è di tipo `long double`, viene chiamata la versione `long double` della funzione;
2. se uno degli argomenti corrispondenti a un parametro generico è di tipo `double` o di un qualsiasi tipo intero, viene chiamata la versione `double` della funzione;
3. negli altri casi viene chiamata la versione `float` della funzione.

La regola numero 2 è un po' insolita: enuncia che un argomento intero provoca la chiamata della versione `double` di una funzione e non la versione `float` come ci si potrebbe aspettare.

 Come esempio assumiamo che siano state dichiarate le seguenti variabili:

```
int i;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
```

Di seguito, per ogni invocazione di macro (colonna sinistra), viene elencata la corrispondente chiamata a funzione (colonna destra):

<i>Invocazione della macro</i>	<i>Equivalente chiamata a funzione</i>
<code>sqrt(i)</code>	<code>sqrt(i)</code>
<code>sqrt(f)</code>	<code>sqrtf(f)</code>
<code>sqrt(d)</code>	<code>sqrt(d)</code>
<code>sqrt(ld)</code>	<code>sqrtl(ld)</code>
<code>sqrt(fc)</code>	<code>csqrtf(fc)</code>
<code>sqrt(dc)</code>	<code>csqrt(dc)</code>
<code>sqrt(ldc)</code>	<code>csqrtl(ldc)</code>

Osservate che scrivere `sqrt(i)` fa sì che venga chiamata la versione `double` della funzione `sqrt` e non la versione `float`.

Queste regole vengono seguite anche dalle macro con più di un parametro. Per esempio, l'invocazione di macro `pow(ld, f)` verrà sostituita dalla chiamata `powl(ld, f)`. Entrambi i parametri della `pow` sono generici, ma visto che uno degli argomenti è di tipo `long double`, la regola 1 impone che venga chiamata la versione `long double` di `pow`.

27.6 L'header `<fenv.h>` (C99): ambiente in virgola mobile

Lo Standard IEEE 754 è quello più usato per la rappresentazione dei numeri a virgola mobile (questo standard è conosciuto anche come IEC 60559, che è il modo

con il quale lo standard C99 vi si riferisce). Lo scopo dell'header `<fenv.h>` è quello di fornire ai programmi l'accesso ai flag di stato floating point e ai modi di controllo specificati dallo standard IEEE. Sebbene questo header fosse stato progettato in un modo assolutamente generale che gli permette di lavorare anche con altre rappresentazioni per la virgola mobile, la ragione della sua creazione è stata il supporto per lo standard IEEE.

Una discussione del perché i programmi abbiano bisogno di accedere ai flag di stato e ai modi di controllo va oltre gli scopi di questo libro. Per avere dei buoni esempi, leggete *"What every computer scientist should know about floating-point arithmetic"* di David Goldberg (ACM Computing Surveys, vol. 23, no. 1: 5–48, Marzo 1991), che è disponibile via Web.

Flag di stato floating point e modi di controllo

La Sezione 7.2 ha trattato di alcune proprietà di base dello Standard IEEE 754. La Sezione 23.4, che ha trattato le aggiunte fatte dal C99 all'header `<math.h>`, ha fornito qualche dettaglio. Una parte di quella discussione, in particolare quella a riguardo delle eccezioni e delle direzioni di arrotondamento, è direttamente rilevante per l'header `<fenv.h>`. Prima di continuare, rivediamo parte del materiale della Sezione 23.4 mentre definiamo alcuni termini nuovi.

Un **flag di stato floating point** è una variabile di sistema che viene impostata quando viene sollevata un'eccezione floating point. Nello standard IEEE, sono presenti cinque tipi di eccezioni floating point: : overflow, underflow, division by zero, invalid operation (il risultato di un'operazione aritmetica è stato NaN) e inexact (il risultato di un'operazione aritmetica ha dovuto essere arrotondato). Ogni eccezione possiede un corrispondente flag di stato.

L'header `<fenv.h>` dichiara un tipo chiamato `fexcept_t` che viene usato per lavorare con i flag di stato floating point. Un oggetto `fexcept_t` rappresenta il valore collettivo di questi flag. Sebbene `fexcept_t` possa essere semplicemente un tipo intero con i singoli bit rappresentanti individualmente i vari flag, lo standard C99 non ne fa un obbligo. Esistono delle alternative, tra cui la possibilità che `fexcept_t` sia una struttura con un membro per ogni eccezione. Questo membro può contenere delle informazioni aggiuntive a riguardo dell'eccezione corrispondente, come l'indirizzo dell'istruzione a virgola mobile che ha sollevato l'eccezione.

Una **modalità di controllo floating point** è una variabile di sistema che può essere impostata dal programma per modificare il comportamento dell'aritmetica a virgola mobile. Lo standard IEEE richiede una modalità di "direzione dell'arrotondamento" che controlla la direzione verso la quale deve essere arrotondato un numero che non può essere espresso in modo esatto usando la rappresentazione a virgola mobile. Ci sono quattro direzioni di arrotondamento: (1) *Arrotondamento verso il più vicino*. Arrotonda verso il più vicino valore rappresentabile. Se un numero cade a metà strada tra due valori, viene arrotondato al valore "pari" (quello il cui bit meno significativo è zero). (2) *Arrotondamento verso lo zero*. (3) *Arrotondamento verso l'infinito positivo*. (4) *Arrotondamento verso l'infinito negativo*. La direzione di arrotondamento di default è quella verso il numero più vicino. Alcune implementazioni dello standard IEEE forniscono due modalità aggiuntive: una modalità che controlla la precisione

dell'arrotondamento e una modalità di "abilitazione delle trappole" che determina se un processore a virgola mobile finisce in trappola (o si ferma) quando viene sollevata un'eccezione.

Il termine **ambiente in virgola mobile** si riferisce alla combinazione dei flag di stato floating point e ai modi di controllo supportati da una particolare implementazione. Un valore di tipo `fenv_t` rappresenta un intero ambiente in virgola mobile. Il tipo `fenv_t`, come il tipo `fexcept_t`, è dichiarato all'interno dell'header `<fenv.h>`.

Macro `<fenv.h>`

È possibile che l'header `<fenv.h>` definisca le macro elencate nella Tabella 27.8, tuttavia solamente due di queste sono obbligatorie (`FE_ALL_EXCEPT` e `FE_DFL_ENV`). Un'implementazione può definire delle macro aggiuntive non presenti nella tabella, i nomi di queste macro devono iniziare per `FE_` e una lettera maiuscola.

Tabella 27.8 Macro presenti in `<fenv.h>`

Nome	Valore	Descrizione
<code>FE_DIVBYZERO</code>	Espessione costante intera i cui bit non si sovrappongono	Definita solo se la corrispondente eccezione floating point è supportata dall'implementazione. Un'implementazione può definire delle macro aggiuntive che rappresentano delle eccezioni floating point.
<code>FE_ALL_EXCEPT</code>	Vedi descrizione	or bitwise di tutte le macro per le eccezioni floating point definite dall'implementazione. Se nessuna di queste macro è definita ha valore 0.
<code>FE_DOWNWARD</code> <code>FE_TONEAREST</code> <code>F_TOWARDZERO</code> <code>FE_UPWARD</code>	Espessione costante intera con valori non negativi distinti	Definita solo se la corrispondente direzione di arrotondamento può essere recuperata e impostata attraverso le funzioni <code>fegetround</code> e <code>fesetround</code> . Un'implementazione può definire delle macro aggiuntive che rappresentano delle direzioni di arrotondamento.
<code>FE_DFL_ENV</code>	Un valore di tipo <code>const fenv_t *</code>	Rappresenta l'ambiente in virgola mobile di default (quello all'avvio del programma). Un'implementazione può definire delle macro aggiuntive che rappresentano degli ambienti a virgola mobile.

Direttiva pragma FENV_ACCESS

L'header `<fenv.h>` fornisce una direttiva [direttiva pragma > 14.5] chiamata `FENV_ACCESS` che viene usata per notificare al compilatore l'intenzione del programma di usare

delle funzioni fornite dall'header stesso. Sapere quali porzioni del programma useranno le capacità offerte dall'header <fenv.h> è importante per il compilatore, perché alcune comuni ottimizzazioni non possono essere eseguite se i modi di controllo non possiedono le loro impostazioni abituali o se possono cambiare durante l'esecuzione del programma.

La direttiva `pragma FENV_ACCESS` si presenta in questo modo:

```
#pragma STDC FENV_ACCESS on-off-switch
```

dove il valore di *on-off-switch* può essere ON, OFF o DEFAULT. Se la direttiva viene usata con il valore ON, informa il compilatore che il programma potrebbe controllare i flag di stato floating point o alterare i modi di controllo floating point. Il valore OFF indica che i flag non verranno analizzati e che è attivo il modo di controllo di default. Il significato del valore DEFAULT è definito dall'implementazione, può rappresentare sia ON che OFF.

La durata di questa direttiva `pragma` dipende dalla posizione in cui viene utilizzata all'interno del programma. Quando si presenta al livello più alto di un file sorgente, al di fuori da qualsiasi dichiarazione esterna, rimane attiva fino alla prossima occorrenza della stessa direttiva o fino alla fine del file. L'unico altro punto dove questa direttiva può presentarsi è l'inizio di un'istruzione composta (eventualmente il corpo di una funzione), in questo caso la direttiva `pragma` rimane attiva fino alla prossima occorrenza della direttiva stessa (anche una interna a un'istruzione composta annidata) o fino alla fine dell'istruzione composta. Alla fine dell'istruzione composta, lo stato della direttiva torna al valore che possedeva prima dell'ingresso nell'istruzione.

È responsabilità del programmatore assicurarsi di usare la direttiva `FENV_ACCESS` per indicare le regioni di un programma nelle quali è necessario un accesso all'hardware a basso livello per le operazioni in virgola mobile. Se un programma analizza dei flag di stato floating point o se viene eseguito con modi di controllo non di default in una regione per la quale la direttiva `pragma` è impostata a OFF, si verifica un comportamento indefinito.

Tipicamente una direttiva `FENV_ACCESS` che specifichi il valore ON viene posta all'inizio del corpo di una funzione:

```
void f(double x, double y)
{
    #pragma STDC FENV_ACCESS ON
}
```

La funzione `f` può analizzare i flag di stato floating point o modificare i modi di controllo quando necessario. Alla fine del corpo di `f`, la direttiva `pragma` ritornerà al suo stato precedente.

Quando un programma, durante l'esecuzione, passa da una regione `FENV_ACCESS "off"` a una regione "on", i flag di stato floating point possiedono dei valori non specificati e i modi di controllo possiedono le loro impostazioni predefinite.

Funzioni per le eccezioni floating point

```

int feclearexcept(int excepts),
int fegetexceptflag(fexcept_t *flgpp, int excepts),
int feraiseexcept(int excepts),
int fesetexceptflag(const fexcept_t *flgpp,
                    int excepts),
int fetestexcept(int excepts),

```

Le funzioni <fenv.h> sono divise in tre gruppi. Le funzioni del primo gruppo hanno a che fare con i flag di stato floating point. Ognuna delle cinque funzioni possiede un parametro int chiamato excepts, il quale è l'or bitwise di una o più macro per le eccezioni floating point (il primo gruppo di macro elencate nella Tabella 27.8). Per esempio, l'argomento passato a una di queste funzioni può essere FE_INVALID | FE_OVERFLOW | FE_UNDERFLOW per rappresentare la combinazione di questi tre flag di stato. L'argomento può anche essere pari a zero per indicare che nessun flag è selezionato. Negli altri casi restituisce un valore diverso da zero.

feclearexcept

La funzione feclearexcept cerca di azzerare le eccezioni floating point rappresentate da excepts. Questa funzione restituisce uno zero se excepts è uguale a zero oppure se tutte le eccezioni specificate sono state azzerate con successo.

fegetexceptflag

La funzione fegetexceptflag cerca di recuperare lo stato dei diversi flag di stato rappresentati da excepts. Questi dati vengono salvati nell'oggetto fexcept_t puntato da flgpp. Questa funzione restituisce uno zero se i flag di stato sono stati salvati con successo, altrimenti restituisce un valore diverso da zero.

feraiseexcept

La funzione feraiseexcept cerca di sollevare le eccezioni floating point supportate che sono rappresentate dall'argomento excepts. Che questa funzione sollevi anche l'eccezione inexact quando solleva quella di overflow o underflow, dipende dall'implementazione (le implementazioni che sono conformi allo standard IEEE possiedono questa proprietà). La funzione feraiseexcept restituisce il valore zero se excepts è uguale a zero oppure se tutte le eccezioni specificate sono state sollevate con successo. Negli altri casi la funzione restituisce un valore diverso da zero.

fesetexceptflag

La funzione fesetexceptflag cerca di impostare i flag di stato rappresentati da excepts. I flag di stato vengono salvati nell'oggetto fexcept_t puntato dall'argomento flgpp. Questo oggetto deve essere stato impostato da una precedente chiamata alla funzione fegetexceptflag. Inoltre il secondo argomento della precedente chiamata alla fegetexceptflag deve aver incluso tutte le eccezioni floating point rappresentate da excepts. Questa funzione restituisce il valore zero se excepts è uguale a zero oppure se tutte le eccezioni specificate sono state impostate con successo. Negli altri casi la funzione restituisce un valore diverso da zero.

fetestexcept

La funzione fetestexcept analizza solo quei flag di stato che sono rappresentati dall'argomento excepts. Questa funzione restituisce l'or bitwise delle macro per le eccezioni floating point corrispondenti ai flag che sono correntemente settati. Per esempio, se il valore di excepts è pari a FE_INVALID | FE_OVERFLOW | FE_UNDERFLOW, la funzione fetestexcept potrebbe restituire il valore FE_INVALID | FE_UNDERFLOW indicando che attualmente, delle eccezioni rappresentate da FE_INVALID, FE_OVERFLOW e FE_UNDERFLOW sono impostati solamente i flag per l'eccezione FE_INVALID e quella FE_UNDERFLOW.

Funzioni per gli arrotondamenti

```
int fegetround(void);
int fesetround(int round);
```

Le funzioni fegetround e fesetround vengono usate per determinare quale sia la direzione di arrotondamento e per modificarla. Entrambe le funzioni si basano sul macro per la direzione di arrotondamento (il terzo gruppo della Tabella 27.8).

- | | |
|------------|---|
| fegetround | La funzione fegetround restituisce il valore della macro che corrisponde alla direzione di arrotondamento corrente. Se la direzione di arrotondamento corrente non può essere determinata o non corrisponde a nessuna delle macro di direzione dell'arrotondamento, la fegetround restituisce un numero negativo. |
| fesetround | Quando le viene passato il valore di una macro di direzione dell'arrotondamento, la funzione fesetround cerca di stabilire la direzione di arrotondamento corrispondente. Se la chiamata va a buon fine, la funzione restituisce il valore zero, altrimenti restituisce un valore diverso da zero. |

Funzioni relative all'ambiente

```
int fegetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
int fesetenv(const fenv_t *envp);
int feupdateenv(const fenv_t *envp);
```

Le ultime quattro funzioni presenti in <fenv.h> hanno a che fare con l'intero ambiente in virgola mobile e non solo con i flag di stato o i modi di controllo. Tutte queste funzioni restituiscono il valore zero se hanno successo nel portare a termine l'azione che viene loro richiesta. In caso contrario restituiscono un valore diverso da zero.

- | | |
|--------------|---|
| fegetenv | La funzione fegetenv cerca di recuperare dal processore l'ambiente in virgola mobile e di salvarlo nell'oggetto puntato dall'argomento envp. |
| feholdexcept | La funzione feholdexcept esegue le seguenti mansioni: (1) salva il corrente ambiente in virgola mobile nell'oggetto puntato da envp, (2) azzerà i flag di stato a virgola mobile, (3) cerca di installare una modalità non-stop (se disponibile) per tutte le eccezioni floating point (in modo che le future eccezioni non causino una trappola o uno stop). |
| fesetenv | La funzione fesetenv cerca di stabilire l'ambiente in virgola mobile rappresentato dall'argomento envp, il quale punta a un ambiente salvato da una precedente chiamata alle funzioni fegetenv o feholdexcept, oppure è uguale a una macro di ambiente in virgola mobile come la FE_DFL_ENV. A differenza della funzione feupdateenv, la fesetenv non solleva nessuna eccezione. Se una chiamata alla fegetenv viene usata per salvare il corrente ambiente in virgola mobile, allora una successiva chiamata alla fesetenv può ripristinarlo nel suo stato precedente. |
| feupdateenv | La funzione feupdateenv cerca di effettuare le seguenti operazioni: (1) salvare le eccezioni floating point correntemente sollevate, (2) installare l'ambiente in virgola mobile puntato da envp e (3) sollevare le eccezioni salvate. L'argomento envp punta a un ambiente in virgola mobile salvato da una precedente chiamata alle funzioni fegetenv o feholdexcept, oppure è uguale a una macro di ambiente in virgola mobile come la FE_DFL_ENV. |

Domande & Risposte

D: Se l'header `<inttypes.h>` include l'header `<stdint.h>`, per quale motivo abbiamo bisogno di quest'ultimo? [p. 737]

R: La ragione principale per l'esistenza di `<stdint.h>` come header separato è quella di permettere ai programmi di un'implementazione freestanding [**implementazione freestanding > 14.3**] di poterlo includere (il C99 richiede alle implementazioni conformi allo standard di fornire l'header `<stdint.h>`, siano esse hosted o freestanding, mentre l'header `<inttypes.h>` è richiesto solo nelle implementazioni hosted). Anche in un ambiente hosted può essere vantaggioso includere `<stdint.h>` invece di `<inttypes.h>` per evitare la definizione di tutte le macro che appartengono a quest'ultimo.

***D:** Nell'header `<math.h>` sono presenti tre versioni della funzione `modf`. Perché non c'è nessuna macro per tipi generici chiamata `modf`? [p. 753]

R: Guardiamo i prototipi per le tre versioni della `modf`

```
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

la `modf` è insolita per il fatto che possiede un parametro puntatore e il suo tipo non è lo stesso nelle tre versioni della funzione (la `frexp` e la `remquo` possiedono un parametro puntatore, ma questo è sempre di tipo `int *`). Avere una macro per tipi generici per la `modf` comporterebbe alcuni problemi difficili da gestire. Per esempio, il significato di `modf(d, &f)`, dove `d` è di tipo `double` ed `f` è di tipo `float` non è chiaro: stiamo chiamando la funzione `modf` o la funzione `modff`? Invece di sviluppare un complicato insieme di regole per una singola funzione (e probabilmente tenendo conto del fatto che la `modf` non è una funzione molto diffusa), il comitato per il C99 ha scelto di non fornire una macro `modf` per tipi generici.

D: Quando una macro `<tgmath.h>` viene invocata con un argomento intero, viene chiamata la versione `double` della funzione corrispondente. In accordo con le normali conversioni aritmetiche non dovrebbe essere chiamata la versione `float` [**normali conversioni aritmetiche > 7.4**]? [p. 754]

R: Stiamo avendo a che fare con una macro e non con una funzione. Di conseguenza le normali conversioni aritmetiche non entrano in gioco. Il comitato per il C99 doveva creare una regola per determinare quale versione di una funzione dovesse essere chiamata quando a una macro `<tgmath.h>` viene passato un argomento intero. Sebbene il comitato in un certo momento considerò che dovesse essere chiamata la versione `float` (per coerenza con le normali conversioni aritmetiche), decise poi che scegliere la versione `double` era la via migliore. Per prima cosa è più sicura: convertire un intero in un `float` può causare una perdita di accuratezza, specialmente per i tipi interi con una dimensione di 32 bit o maggiore. Secondariamente, questa scelta presenta meno sorprese al programmatore. Supponete che `i` sia una variabile intera. Se l'header `<tgmath.h>` non è incluso, la chiamata `sin(i)` invoca la funzione `sin`. D'altra parte se l'header `<tgmath.h>` è incluso, la stessa chiamata invoca la macro `sin` e, visto che `i` è un intero, il preprocessore sostituisce la macro con la funzione `sin`. Alla fine il risultato è lo stesso.

D: Quando un programma invoca una delle macro per tipi generici presenti in `<tgmath.h>`, come fa l'implementazione a determinare quale funzione chiamare? C'è un modo per una macro di analizzare il tipo dei suoi argomenti?

R: Un aspetto insolito di `<tgmath.h>` è che le sue macro necessitano di essere in grado di analizzare il tipo degli argomenti che vengono loro passati. Il C non ha nessuna funzionalità per analizzare i tipi e quindi normalmente sarebbe impossibile scrivere una macro di questo tipo. Le macro `<tgmath.h>` si basano su particolari funzionalità fornite da un particolare compilatore al fine di rendere possibile questa analisi. Non sappiamo cosa siano queste funzionalità e non abbiamo la garanzia che siano portabili da un compilatore all'altro.

Esercizi

Sezione 27.1



- Individuate le dichiarazioni dei tipi `intN_t` e `uintN_t` nell'header installato sul vostro sistema. Quali valori di N sono supportati?
- Scrivete le macro parametriche `NT32_C(n)`, `UINT32_C(n)`, `INT64_C(n)` e `UINT64_C(n)`, assumendo che i tipi `int` e `long int` abbiano una dimensione di 32 bit e che il tipo `long long int` abbia una dimensione di 64 bit. *Suggerimento:* usate l'operatore del preprocessore `##` per aggiungere un suffisso a `n` contenente una combinazione dei caratteri `L` e/o `U` (leggete la Sezione 7.1 per una discussione su come usare i suffissi `L` e `U` con le costanti intere).

Sezione 27.2



- In ognuna delle istruzioni seguenti assumete che la variabile `i` abbia il tipo originale indicato. Usando le macro dell'header `<inttypes.h>`, modificate ogni istruzione in modo che funzioni correttamente anche se il tipo di `i` viene modificato in quello nuovo indicato.

(a) <code>printf("%d", i);</code>	Tipo originale: <code>int</code>	Nuovo tipo: <code>int8_t</code>
(b) <code>printf("%12.4d", i);</code>	Tipo originale: <code>int</code>	Nuovo tipo: <code>int32_t</code>
(c) <code>printf("%-6o", i);</code>	Tipo originale: <code>unsigned int</code>	Nuovo tipo: <code>uint16_t</code>
(d) <code>printf("%#x", i);</code>	Tipo originale: <code>unsigned int</code>	Nuovo tipo: <code>uint64_t</code>

Sezione 27.5



- Assumete che siano valide le seguenti dichiarazioni di variabili:

```
int i;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
```

Ognuna delle seguenti è un'invocazione di una macro appartenente all'header `<tgmath.h>`. Mostrate come si presenterà l'invocazione dopo il preprocessamento,

una volta che la macro è stata sostituita da una funzione appartenente all'header `<math.h>` o all'header `<complex.h>`.

- (a) `tan(i)`
- (b) `fabs(f)`
- (c) `asin(d)`
- (d) `exp(ld)`
- (e) `log(fc)`
- (f) `acosh(dc)`
- (g) `nexttoward(d, ld)`
- (h) `remainder(f, i)`
- (i) `copysign(d, ld)`
- (j) `carg(i)`
- (k) `cimag(f)`
- (l) `conj(ldc)`

Progetti di programmazione

C99

1. Apportate le seguenti modifiche al programma `quadratic.c` della Sezione 27.4.
 - (a) Fate in modo che l'utente immetta i coefficienti del polinomio (i valori delle variabili `a`, `b` e `c`).
 - (b) Fate in modo che il programma controlli il discriminante prima di visualizzare i valori delle radici. Se il discriminante è negativo, fate in modo che il programma visualizzi le radici nel modo originale. Se invece non è negativo, fate in modo che il programma visualizzi le radici come numeri reali (senza parte immaginaria). Per esempio, se l'equazione quadratica è $x^2 + x - 2 = 0$, l'output del programma sarà

```
root1 = 1
root2 = -2
```

- (c) Modificate il programma in modo che visualizzi un numero complesso con una parte immaginaria negativa come $a - bi$ invece che $a + -bi$. Per esempio, l'output del programma con i coefficienti originali sarà

```
root1 = -0.2 + 0.4i
root2 = -0.2 - 0.4i
```

C99

2. Scrivete un programma che converte un numero complesso in coordinate cartesiane a partire dalla forma polare. L'utente immetterà a e b (la parte reale e quella immaginaria del numero), il programma visualizzerà i valori di r e θ .

C99

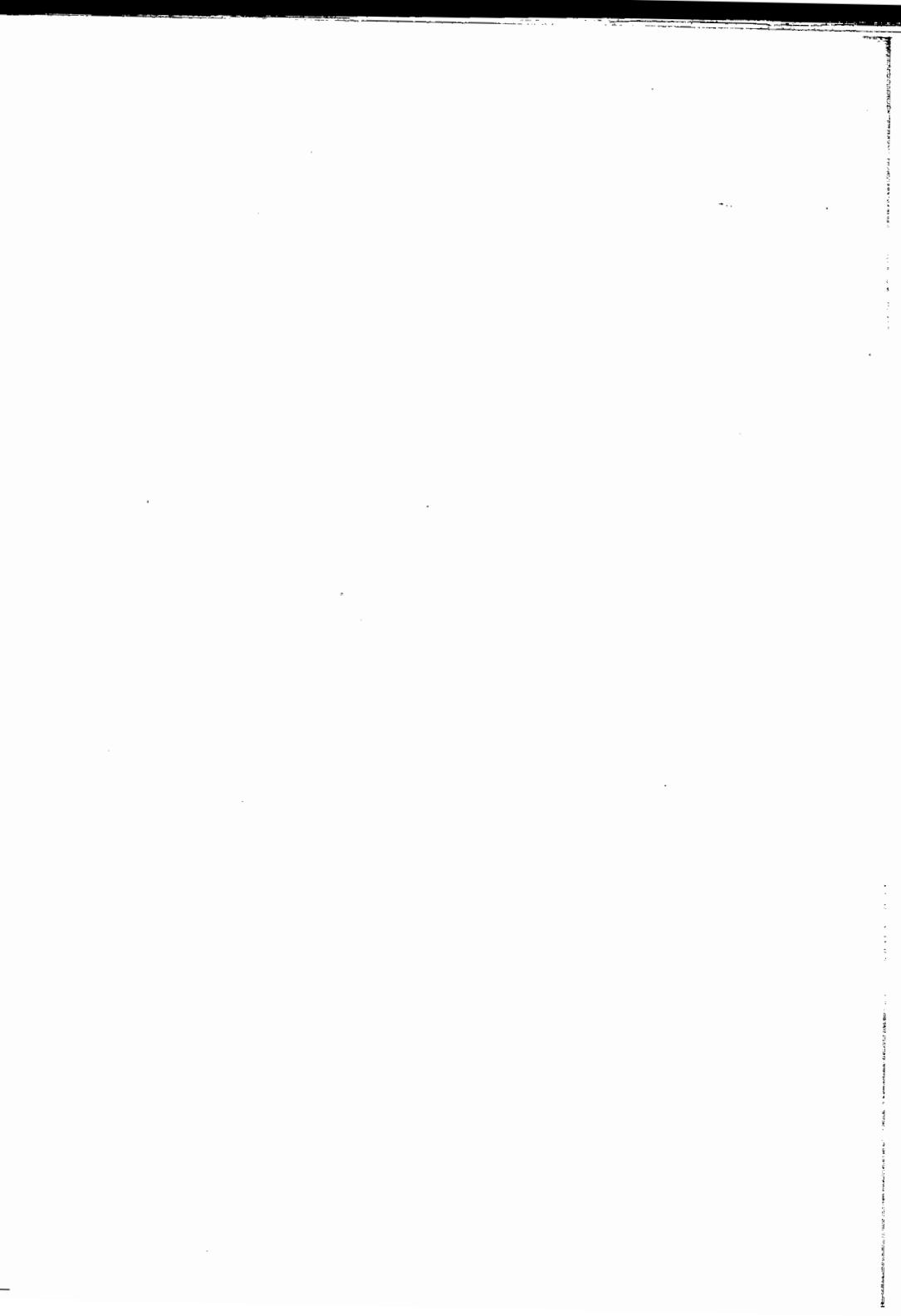
3. Scrivete un programma che converte un numero complesso in coordinate polari a partire dalla forma cartesiana. Dopo che l'utente ha immesso i valori di r e θ , programma dovrà visualizzare il numero nella forma $a + bi$, dove

$$a = r \cos \theta$$

$$b = r \sin \theta$$

C99

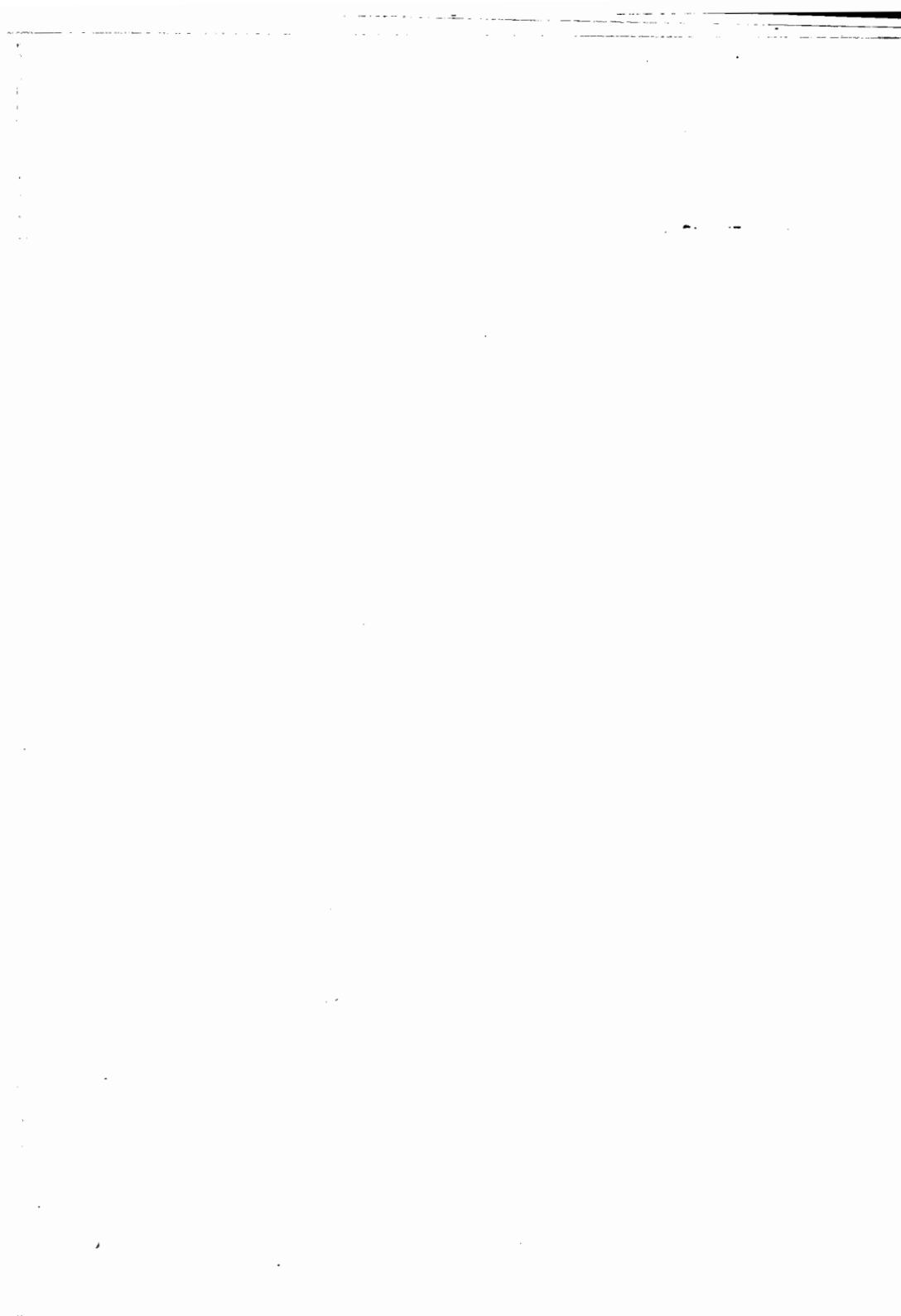
4. Scrivete un programma che visualizzi le radici n -esime dell'unità quando viene dato un intero positivo n . Le radici n -esime sono date dalla formula $e^{2\pi ik/n}$, dove k è un intero compreso tra 0 e $n-1$.



Appendice A

Operatori C

Precedenza	Nome	Simboli/i	Associatività
1	Individuazione vettori	[]	Sinistra
1	Chiamata a funzione	()	Sinistra
1	Membro unione e struttura	. ->	Sinistra
1	Incremento (prefisso)	++	Sinistra
1	Decremento (prefisso)	--	Sinistra
2	Incremento (prefisso)	++	Destra
2	Decremento (prefisso)	--	Destra
2	Indirizzo	&	Destra
2	Indirection	*	Destra
2	Più unario	+	Destra
2	Meno unario	-	Destra
2	Complemento bitwise	~	Destra
2	Negazione logica	!	Destra
2	Dimensione	sizeof	Destra
3	Cast	()	Destra
4	Moltiplicativi	* / %	Sinistra
5	Additivi	+ -	Sinistra
6	Scorrimento bitwise	<< >>	Sinistra
7	Relazionali	< > <= >=	Sinistra
8	Uguaglianza	== !=	Sinistra
9	and bitwise	&	Sinistra
10	or esclusivo bitwise	^	Sinistra
11	or inclusivo bitwise		Sinistra
12	and logico	&&	Sinistra
13	or logico		Sinistra
14	Condizionale	: :	Destra
15	Assegnamento	= *= /= %= += -= <<= >>= &= ^= =	Destra
16	Virgola	,	Sinistra



Appendice B

C99 e C89 a confronto

Questa appendice elenca molte delle maggiori differenze presenti tra il C89 e il C99 (le differenze minori sono troppo numerose per essere elencate qui). I titoli indicano quale capitolo contenga la trattazione principale di ogni caratteristica del C99. Alcune modifiche attribuite al C99 in effetti sono avvenute prima, nell'Amendment 1 dello standard C89. Queste modifiche sono indicate con "Amendment 1".

2 Fondamenti del C

// commenti Il C99 aggiunge un secondo tipo di commenti che iniziano con `//`.

identificatori Il C89 richiede che il compilatore ricordi i primi 31 caratteri degli identificatori. Il C99 richiede che vengano ricordati 63 caratteri. Nel C89 solamente i primi sei caratteri dei nomi con collegamento esterno sono significativi. Inoltre, il case delle lettere può anche non contare. Nel C99 sono significativi i primi 31 caratteri e il case delle lettere viene tenuto in considerazione.

keyword Nel C99 sono state aggiunte cinque nuove keyword: `inline`, `restrict`, `_Bool`, `_Complex` e `_Imaginary`.

valori restituiti dal main Nel C89, se un programma raggiunge la fine della funzione `main` senza eseguire l'istruzione `return`, il valore restituito al sistema operativo non è definito. Nel C99, invece, se è stato dichiarato che il `main` restituisce un `int`, il programma restituisce il valore 0 al sistema operativo.

4 Espressioni

operatori / e % Il C89 afferma che se uno degli operandi è negativo, il risultato di una divisione intera può essere arrotondato per eccesso o per difetto. Inoltre, se `i` o `j` è negativo, il segno di `i % j` dipende dall'implementazione. Nel C99, il risultato di una divisione viene sempre troncato verso lo zero e il valore di `i % j` ha sempre lo stesso segno di `i`.

5 Istruzioni di selezione

tipo _Bool Il C99 fornisce un tipo booleano chiamato `_Bool`. Il C89 non ha alcun tipo booleano.

6 Cicli

cicli for Nel C99 la prima espressione di un'istruzione for può essere sostituita da una dichiarazione, permettendo all'istruzione di dichiarare la sua variabile/i di controllo.

7 Tipi base

tipi interi long long Il C99 fornisce due tipi interi standard aggiuntivi: `long long int` e `unsigned long long int`.

tipi interi estesi Oltre ai tipi interi standard, il C99 permette dei tipi interi estesi (con e senza segno) definiti dall'implementazione.

costanti intere long long Il C99 fornisce una modo per indicare che una costante intera sia di tipo `long long int` o `unsigned long long int`.

tipo delle costanti intere Le regole del C99 per determinare il tipo di una costante intera sono diverse da quelle del C89.

costanti in virgola mobile esadecimale Il C99 fornisce un modo per scrivere in esadecimale le costanti in virgola mobile.

conversioni implicite Le regole per le conversioni implicite del C99 sono in qualche modo diverse da quelle del C89. Questo avviene principalmente a causa dei tipi base aggiuntivi del C99.

8 Vettori

designated initializer Il C99 supporta i designated initializer, i quali possono essere usati per inizializzare i vettori, le strutture e le unioni.

vettori a lunghezza variabile Nel C99 la lunghezza di un vettore può essere specificata da un'espressione che non è costata, ammesso che il vettore non abbia un durata di memorizzazione statica e che la sua dichiarazione non contenga un inizializzatore.

9 Funzioni

nessun tipo restituito di default Se nel C89 il tipo restituito da una funzione viene omesso, si presume che la funzione restituisca un valore di tipo `int`. Nel C99 invece non è ammesso omettere il tipo restituito di una funzione.

dichiarazioni e istruzioni mischiate Nel C89 le dichiarazioni devono precedere le istruzioni presenti all'interno di un blocco (incluso il corpo di una funzione). Nel C99 le

dichiarazioni e le istruzioni possono essere meschiate, fintanto che ogni variabile viene dichiarata precedentemente della prima istruzione che utilizza la variabile stessa.

dichiarazioni o definizioni richieste prima di una chiamata a funzione Il C99 richiede che una dichiarazione o una definizione di una funzione siano presenti prima di ogni chiamata alla funzione stessa. Il C89 non possiede questo obbligo: se una funzione viene chiamata senza una precedente dichiarazione o definizione, il compilatore assume che la funzione restituisca un valore `int`.

parametri costituiti da vettori a lunghezza variabile Il C99 permette di utilizzare un vettore a lunghezza variabile come parametro. Nella dichiarazione della funzione il simbolo `*` si presenta all'interno delle parentesi quadre per indicare che il parametro è costituito da un vettore a lunghezza variabile.

parametri costituiti da vettori static Il C99 ammette l'uso della parola `static` nella dichiarazione di un parametro costituito da un vettore, indicando una lunghezza minima per la prima dimensione del vettore.

letterali composti Il C99 supporta l'uso di letterali composti, i quali permettono la creazione di valori per vettori e strutture privi di nome.

dichiarazione del main Il C99 permette che il `main` venga dichiarato in un modo definito dall'implementazione, con un tipo restituito diverso da `int` e/o parametri diversi da quelli specificati dallo standard.

istruzione return senza espressione Nel C89, eseguire l'istruzione `return` senza un'espressione all'interno di una funzione non-void provoca un comportamento indefinito (ma solo se il programma cerca di usare il valore restituito dalla funzione). Nel C99 un'istruzione di questo tipo non è ammessa.

14 Preprocessore

macro aggiuntive predefinite Il C99 fornisce diverse nuove macro predefinite.

argomenti vuoti per le macro Il C99 permette che uno o tutti gli argomenti di una macro siano vuoti, ammesso che la chiamata contenga il numero corretto di virgolette.

macro con un numero variabile di argomenti Nel C89 una macro deve possedere un numero fissato di argomenti (se ne ha). Il C99 permette che le macro accettino un numero illimitato di argomenti.

identificatore __func__ Nel C99 l'identificatore `__func__` si comporta come una variabile stringa che contiene il nome della funzione correntemente in esecuzione.

direttive pragma standard Nel C89 non sono presenti delle direttive `pragma standard`. Il C99 ne possiede tre: `CX_LIMITED_RANGE`, `FENV_ACCESS`, e `FP_CONTRACT`.

operatore _Pragma Il C99 fornisce l'operatore `_Pragma`, il quale viene usato congiuntamente alla direttiva `#pragma`.

16 Strutture, unioni ed enumerazioni

compatibilità dei tipi struttura Nel C89 strutture definite in file diversi sono compatibili se i loro membri hanno gli stessi nomi e compaiono nello stesso ordine, con i membri corrispondenti che sono di tipo compatibile. Il C99 richiede anche che entrambe le strutture abbiano lo stesso tag o nessuna delle due abbia un tag.

virgola trascinata nelle enumerazioni Nel C99 l'ultima costante di un'enumerazione può essere seguita da una virgola.

17 Uso avanzato dei puntatori

restricted pointer Il C99 possiede una nuova keyword (`restrict`) che può comparire nella dichiarazione di un puntatore.

membri vettore flessibili Il C99 permette che l'ultimo membro di una struttura sia un vettore di lunghezza non specificata.

18 Dichiarazioni

scope di blocco per le istruzioni di selezione e di iterazione Nel C99 le istruzioni di selezione (`if` e `switch`) e le istruzioni di iterazione (`while`, `do` e `for`), assieme con le istruzioni "interne" che controllano, sono considerate come dei blocchi.

inizializzatori per vettori, strutture e unioni Nel C89 un inizializzatore racchiuso da parentesi graffe per un vettore, una struttura o un'unione deve contenere solamente espressioni costanti. Nel C99 questa restrizione si applica solo se la variabile ha durata di memorizzazione statica.

funzioni inline Il C99 permette alle funzioni di essere dichiarate `inline`.

21 La libreria standard

header <stdbool.h> Il C99 introduce l'header `<stdbool.h>`, che definisce le macro `bool`, `true` e `false`.

22 Input/Output

specifiche di conversione ...printf Nel C99 le specifiche di conversione per le funzioni `...printf` hanno subito un certo numero di modifiche, con nuovi modificatori di lunghezza, nuovi specicatori di conversione, la possibilità di scrivere infinito e NaN, e il supporto per i wide character. Anche le conversioni `%le`, `%lE`, `%lf`, `%lg` e `%lG`, che nel C89 provocavano un comportamento indefinito, sono ammesse nel C99.

specifiche di conversione ...scanf Nel C99 le specifiche di conversione per le funzioni `...scanf` possiedono dei nuovi modificatori di lunghezza, nuove specifiche di conversione, la capacità di leggere i valori infinito e NaN, e il supporto per i wide character.

funzione snprintf Il C99 aggiunge la funzione `snprintf` all'header `<stdio.h>`.

23 Supporto di libreria per i dati numerici e i caratteri.

macro aggiuntive nell'header <float.h> Il C99 aggiunge all'header <float.h> le macro DECIMAL_DIG e FLT_EVAL_METHOD.

macro aggiuntive all'header <limits.h> Nel C99 l'header <limits.h> contiene tre nuove macro che descrivono le caratteristiche dei tipi long long int.

macro math_errhandling Il C99 dà alle implementazioni la scelta di come informare il programma che si è verificato un errore in una funzione matematica: attraverso un valore salvato nella variabile errno, attraverso un'eccezione floating point o con entrambi i metodi. Il valore della macro math_errhandling (definita in <math.h>) indica come gli errori vengono segnalati in una particolare implementazione.

funzioni aggiuntive nell'header <math.h> Il C99 aggiunge due nuove versioni per la maggior parte delle funzioni <math.h>, una per il tipo float e una per il tipo long double. Il C99 aggiunge a <math.h> anche un certo numero di funzioni completamente nuove e macro parametriche.

24 Gestione degli errori

macro EILSEQ Il C99 aggiunge all'header <errno.h> la macro EILSEQ.

25 Caratteristiche per l'internazionalizzazione

digrafi I digrafi, che sono dei simboli costituiti da due caratteri che possono essere usati in sostituzione ai token ,], {, }, # e ##, sono stati introdotti dal C99 (Amendment 1).

header <iso646.h> L'header <iso646.h>, che definisce delle macro che rappresentano gli operatori contenenti i caratteri &, |, ~, ! e ^, è stato introdotto dal C99 (Amendment 1).

universal character name Il C99 introduce i nomi universali per i caratteri, che forniscono un modo per incorporare i caratteri UCS nel codice sorgente di un programma.

header <wchar.h> L'header <wchar.h>, che fornisce delle funzioni per l'input/output dei wide character e la manipolazione delle stringhe wide, è stato introdotto dal C99 (Amendment 1).

header <wctype.h> L'header <wctype.h>, ovvero la versione wide character dell'header <ctype.h>, è stato introdotto dal C99. Questo header fornisce le funzioni per la classificazione e la modifica del case dei wide character (Amendment 1).

26 Funzioni di libreria varie

macro va_copy Il C99 aggiunge all'header <stdarg.h> una macro parametrica chiamata va_copy.

funzioni aggiuntive nell'header <stdio.h> Il C99 aggiunge all'header <stdio.h> le funzioni vsnprintf, vfscanf, vscanf e vsscanf.

funzioni aggiuntive nell'header <stdlib.h> Il C99 aggiunge nell'header `<stdlib.h>` cinque funzioni di conversione, la funzione `_Exit`, e le versioni `long long` per le funzioni `abs` e `div`.

specificatori di conversione aggiuntivi per la strftime Il C99 aggiunge un certo numero di nuovi specificatori di conversione per la `strftime`. Permette anche di usare i caratteri `E` o `0` per modificare il significato di certi specificatori di conversione.

27 Supporto aggiuntivo del C99 per la matematica

header <stdint.h> Il C99 introduce l'header `<stdint.h>`, che dichiara i tipi interi con dimensione specificata.

header <inttypes.h> Il C99 introduce l'header `<inttypes.h>`, che fornisce delle macro che sono utili per l'input/output dei tipi interi dichiarati in `<stdint.h>`.

tipi complessi Il C99 fornisce tre tipi complessi: `float _Complex`, `double _Complex` e `long double _Complex`.

header <complex.h> Il C99 introduce l'header `<complex.h>` che fornisce delle funzioni per eseguire operazioni matematiche sui numeri complessi.

header <tgmath.h> Il C99 introduce l'header `<tgmath.h>` che fornisce delle macro per tipi generici che facilitano le chiamate alle funzioni di libreria presenti in `<math.h>` e `<complex.h>`.

header <fenv.h> Il C99 introduce l'header `<fenv.h>`, che fornisce ai programmi accesso ai flag di stato floating point e ai modi di controllo.

Appendice C

C89 e K&R C a confronto

Questa appendice elenca le differenze più significative tra il C89 e il K&R C (il linguaggio descritto nella prima edizione del libro di Kernighan e Ritchie chiamato *The C Programming Language*). I titoli indicano quale capitolo di questo libro ha discusso ognuna delle caratteristiche del C89. Questa appendice non riguarda la libreria del C, che è molto cambiata nel corso degli anni. Per le altre (meno importanti) differenze esistenti tra il C89 e il K&R C, consultate le Appendici A e C della seconda edizione di K&R.

La maggior parte dei compilatori C gestiscono tutti il C89, ma questa appendice è utile nel caso vi capitasse di incontrare dei vecchi programmi che sono stati originalmente scritti per i compilatori pre-C89.

2 Fondamenti del C

identificatori Nel K&R C solamente i primi otto caratteri di un identificatore sono significativi.

keyword Il K&R C è privo delle keyword `const`, `enum`, `signed`, `void` e `volatile`. Nel K&R C la parola `entry` è una keyword.

4 Espressioni

+ unario Il K&R C non supporta l'operatore `+ unario`

5 Istruzioni di selezione

switch Nel K&R C l'espressione di controllo (e le label dei casi) di un costrutto `switch` sono di tipi `int` dopo una promozione. Nel C89, le espressioni e le label possono essere un qualsiasi tipo integrale, inclusi `unsigned int` e `long int`.

7 Tipi base

tipi senza segno Il K&R C fornisce solamente un tipo senza segno (`unsigned int`).

signed Il K&R C non supporta lo specificatore di tipo **signed**.

suffissi dei numeri Il K&R C non supporta il suffisso **U** (o **u**) per specificare che una costante intera è di tipo senza segno e non supporta nemmeno il suffisso **F** (o **f**) per indicare che una costante in virgola mobile debba essere memorizzata come un valore **float** invece che come valore **double**. Nel K&R C il suffisso **L** (o **l**) non può essere usato con le costanti a virgola mobile.

long float Il K&R C permette l'uso di **long float** come sinonimo di **double**. Questo utilizzo non è ammesso dal C89.

long double Il K&R C non supporta il tipo **long double**.

sequenze di escape Le sequenze di escape non esistono nel K&R C. Inoltre il K&R C non supporta le sequenze di escape esadecimali.

size_t Nel K&R C l'operatore **sizeof** restituisce un valore di tipo **int**. Nel C89 restituisce un valore di tipo **size_t**.

normali conversioni aritmetiche Il K&R C richiede che gli operandi **float** vengano convertiti in **double**. Specifica inoltre che combinare un intero senza segno più piccolo con un intero con segno più grande produce sempre un risultato senza segno.

9 Funzioni

definizione di funzioni In una definizione di funzione C89, i tipi dei parametri sono inclusi nell'elenco dei parametri:

```
double square(double x)
{
    return x * x;
}
```

Il K&R C richiede che i tipi dei parametri vengano specificati in un elenco separato:

```
double square(x)
double x;
{
    return x * x;
}
```

Una dichiarazione di funzione C89 (prototipo) specifica i tipi dei parametri delle funzioni (e parimenti anche i nomi, se lo si desidera):

```
double square(double x);
double square(double); /* forma alternativa */
int rand(void); /* nessun parametro */
```

Una dichiarazione di funzione K&R C omette tutte le informazioni riguardanti i parametri:

```
double square();
int rand();
```

chiamate a funzione Quando viene usata una definizione o una dichiarazione C, il compilatore non controlla che la funzione venga chiamata con un numero appropriato di argomenti e che questi siano del tipo giusto. Inoltre gli argomenti non vengono convertiti automaticamente ai tipi dei corrispondenti parametri. Si applicano invece promozioni integrali e gli argomenti float vengono convertiti al tipo double.

void Il K&R C non supporta il tipo void.

12 Puntatori e vettori

sottrazione dei puntatori Nel K&R C, sottrarre due puntatori produce un valore intero mentre nel C89 produce un valore di tipo ptrdiff_t.

13 Stringhe

stringhe letterali Nel K&R C delle stringhe letterali adiacenti non vengono concatenate. Inoltre il K&R C non proibisce la modifica di stringhe letterali.

inizializzazione delle stringhe Nel K&R C un inizializzatore per un vettore di dimensione n è limitato a $n - 1$ caratteri (lasciando spazio per il carattere null alla fine). Il C89 permette all'inizializzatore di avere una lunghezza pari a n .

14 Il preprocessore

#elif, #error, #pragma Il K&R C non supporta le direttive #elif, #error e #pragma.
#, ##, defined Il K&R C non supporta gli operatori #, ## e defined.

16 Strutture, unioni ed enumerazioni

membri e tag di strutture e unioni Nel C89 ogni struttura e unione possiede il suo spazio dei nomi per i membri, i tag delle strutture e delle unioni sono mantenuti in uno spazio dei nomi separato. Il K&R C utilizza un singolo nome per i membri e i tag, in conseguenza non possono avere lo stesso nome (con qualche eccezione), e i membri e i tag non possono sovrapporsi..

operazioni sull'intera struttura Il K&R C non permette alle strutture di essere operate su di esse come si fa con le unioni.

enumerazioni Il K&R C non supporta le enumerazioni.

17 Uso avanzato dei puntatori

void * Nel C89, void * viene usato come un tipo puntatore "generico", la funzione malloc restituisce un valore di tipo void *. Nel K&R C a questo scopo viene usato il tipo char *.

mescolanza di puntatori Il K&R C permette a puntatori di tipi diversi di venir mischiati in assegnamenti e confronti. Nel C89, i puntatori di tipo `void *` possono essere mischiati con puntatoti di altri tipi, ma qualsiasi altra mescolanza non è permessa senza un casting. Similmente, il K&R C permette la mescolanza di interi e puntatori nelle assegnazioni e nei confronti, mentre il C89 richiede il casting.

puntatori a funzione Se `pf` è un puntatore a una funzione, il C89 per invocare la funzione permette di utilizzare sia la forma `(*pf)()` sia la forma `pf()`. Il K&R C permette solo la forma `(*pf)()`.

18 Dichiarazioni

const e volatile Il K&R C non supporta i qualificatori di tipo `const` e `volatile`.

inizializzazione di vettori, strutture e unioni Il K&R C non permette l'inizializzazione automatica di vettori e strutture e nemmeno permette l'inizializzazione delle unioni (indipendentemente dalla durata di memorizzazione).

25 Caratteristiche per l'internazionalizzazione

wide character Il K&R C non supporta le costanti `wide character` e le stringhe letterali `wide`.

sequenze trigrafiche Il K&R C non supporta le sequenze trigrafiche.

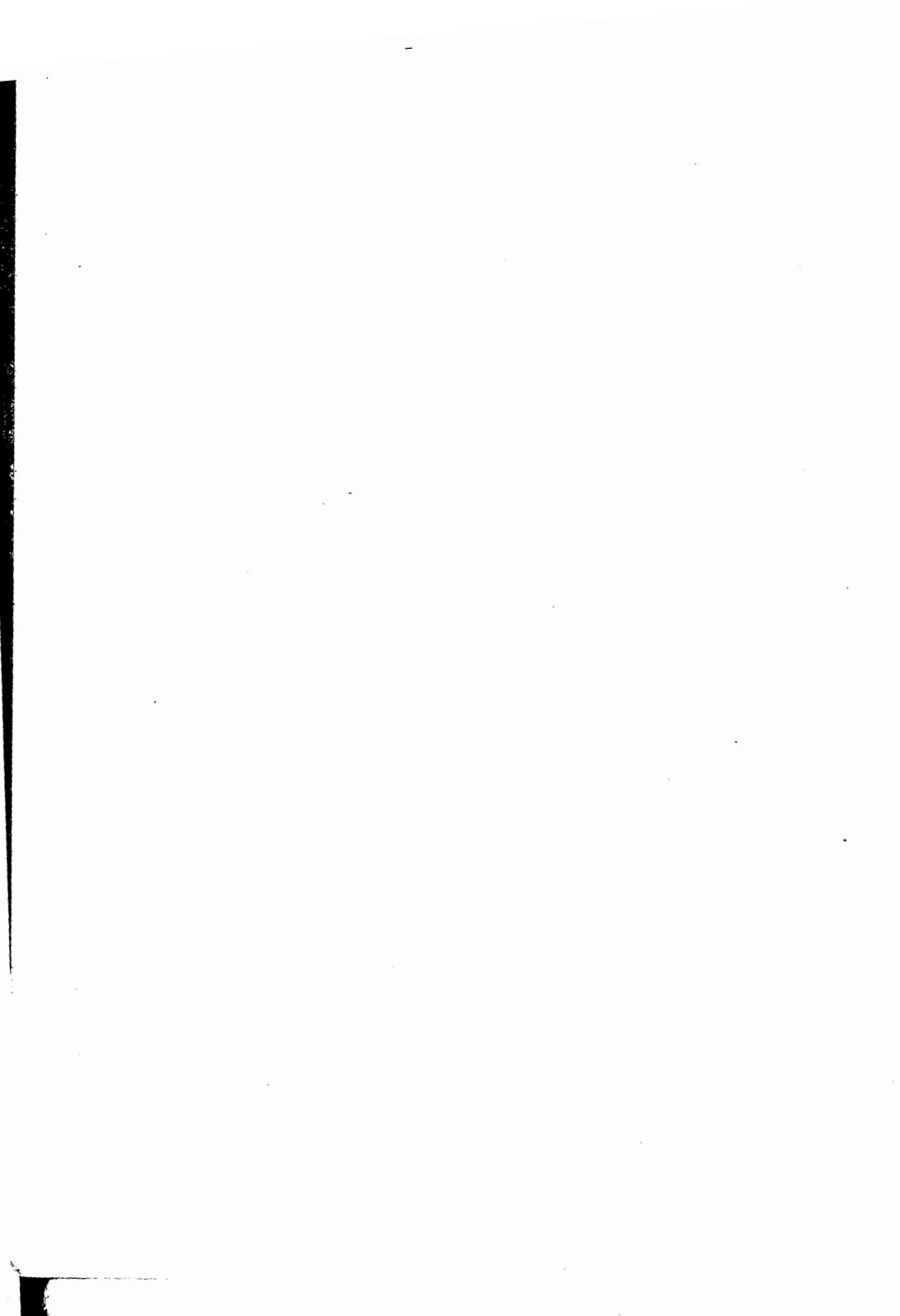
26 Funzioni di libreria varie

argomenti variabili Il K&R C non fornisce un modo portabile per scrivere funzioni con un numero variabile di argomenti ed è privo della notazione ... (ellissi).

Appendice D

Set di caratteri ASCII

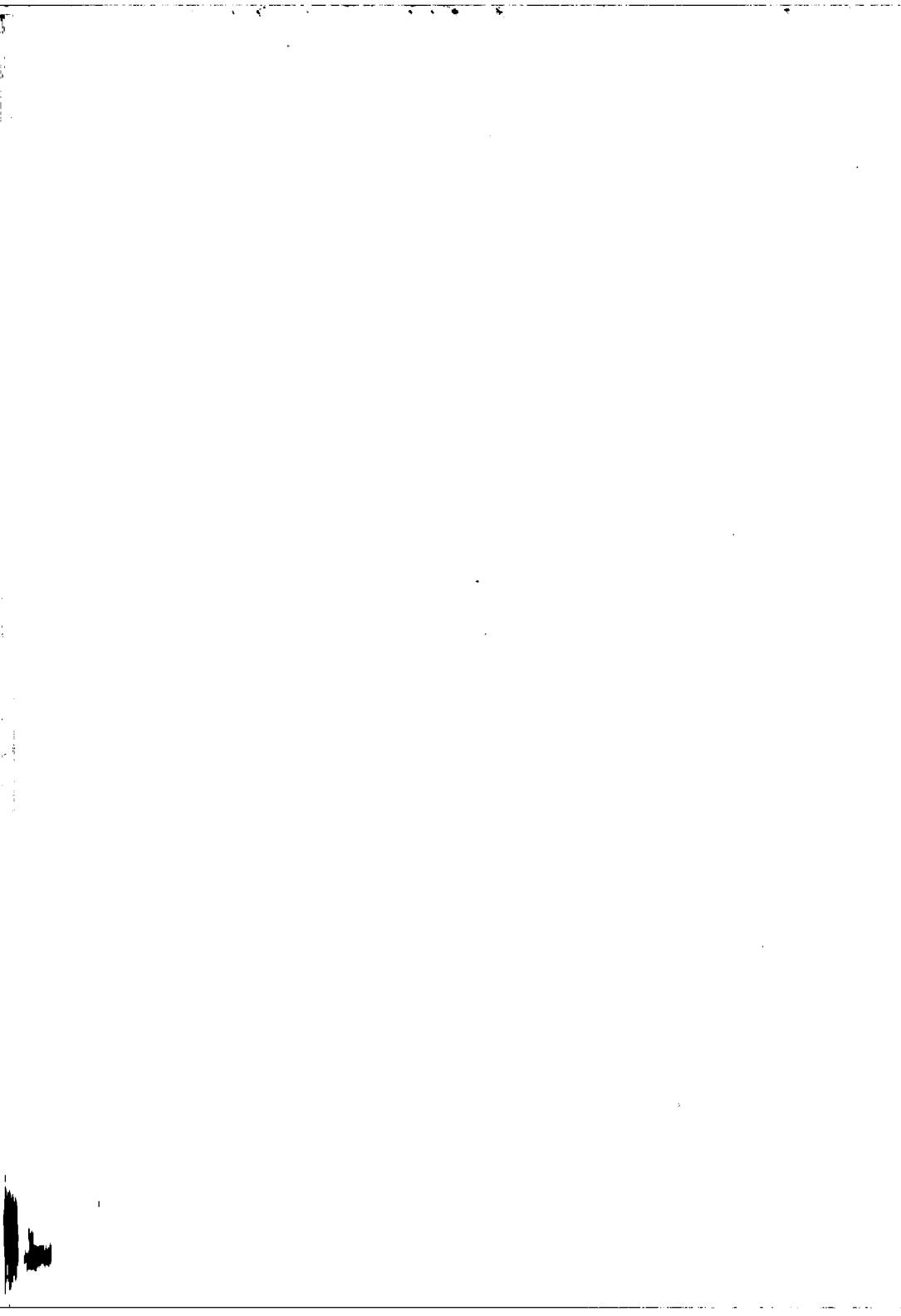
Escape Sequence							
Decimal	Oct	Hex	Char	Character			
0	\0	\x00		nul		32	@
1	\1	\x01		soh	(^A)	33 !	97 a
2	\2	\x02		stx	(^B)	34 "	98 b
3	\3	\x03		etx	(^C)	35 #	99 c
4	\4	\x04		eot	(^D)	36 \$	100 d
5	\5	\x05		erq	(^E)	37 %	101 e
6	\6	\x06		ack	(^F)	38 &	102 f
7	\7	\x07	\a	bel	(^G)	39 ,	103 g
8	\10	\x08	\b	bs	(^H)	40 (104 h
9	\11	\x09	\t	ht	(^I)	41)	105 i
10	\12	\x0a	\n	lf	(^J)	42 *	106 j
11	\13	\x0b	\v	vt	(^K)	43 +	107 k
12	\14	\x0c	\f	ff	(^L)	44 ,	108 l
13	\15	\x0d	\r	cr	(^M)	45 -	109 m
14	\16	\x0e		so	(^N)	46 .	110 n
15	\17	\x0f		si	(^O)	47 /	111 o
16	\20	\x10		dle	(^P)	48 0	112 p
17	\21	\x11		dc1	(^Q)	49 1	113 q
18	\22	\x12		dc2	(^R)	50 2	114 r
19	\23	\x13		dc3	(^S)	51 3	115 s
20	\24	\x14		dc4	(^T)	52 4	116 t
21	\25	\x15		nak	(^U)	53 5	117 u
22	\26	\x16		syn	(^V)	54 6	118 v
23	\27	\x17		etb	(^W)	55 7	119 w
24	\30	\x18		can	(^X)	56 8	120 x
25	\31	\x19		em	(^Y)	57 9	121 y
26	\32	\x1a		sub	(^Z)	58 :	122 z
27	\33	\x1b		esc		59 ;	123 {
28	\34	\x1c		fs		60 <	124 \
29	\35	\x1d		gs		61 =	125 }
30	\36	\x1e		rs		62 >	126 ^
31	\37	\x1f		us		63 ?	127 ~ del



Bibliografia

Programmazione C

- Feuer, A.R., *The C Puzzle Book*, Revised Printing, Addison-Wesley, Reading, Mass., 1989. Contiene numerosi "puzzle", ovvero piccoli programmi C dei quali viene chiesto al lettore di predire l'output. Il libro illustra l'output corretto per ogni programma e fornisce una spiegazione dettagliata di come questo funzioni. È un buon libro per approfondire la vostra conoscenza del C e per rivedere le sottigliezze del linguaggio.
- Harbison, S.P., III, e G.L. Steele, Jr., *C: A Reference Manual*, Fifth Edition, Prentice-Hall, Upper Saddle River, N.J., 2002. La guida completa di riferimento per il C, una lettura essenziale per chi vuol diventare un esperto di questo linguaggio. Tratta con particolare dettaglio sia il C89 che il C99, con frequenti discussioni sulle differenze di implementazione che si riscontrano nei compilatori C. Non è un tutorial: assume che il lettore abbia già una buona conoscenza del C. La seconda edizione riflette le modifiche apportate nel C89.
- Koenig, A., *C Traps and Pitfalls*, Addison-Wesley, Reading, Mass., 1989. Un eccellente compendio sugli errori comuni (e di alcuni non comuni) che vengono fatti quando l'uomo avvisato mezzo salvato.
- Plauger, P.J., *The Standard C Library*, Prentice-Hall, Englewood Cliffs, N.J., 1992. Non spiega solamente tutti gli aspetti della libreria standard del C89, ma fornisce una risposta completa! Non c'è modo migliore per imparare la libreria che studiare su questo libro. Anche se il vostro interesse per la libreria è minimo, il libro è degno di essere acquistato per avere l'opportunità di studiare il codice C scritto da un maestro.
- Ritchie, D.M., *The development of the C programming language* in *History of Programming Languages II*, edito da T. J. Bergin, Jr. e R. G. Gibson, Jr., Addison-Wesley, Reading, Mass., 1996, pagine 671-687. La storia del C in breve scritta dal progettista del linguaggio, per la seconda conferenza ACM SIGPLAN sulla storia dei linguaggi di programmazione, tenutasi nel 1993. L'articolo è seguito dalle trascrizioni della presentazione tenuta da Ritchie alla conferenza e della sessione domande-e-risposte avvenuta con il pubblico.



Indice analitico

A

algoritmo Quicksort, 213
alias, 256
allocazione dinamica sulla memoria, 428
ambiente in virgola mobile, 756
American National Standard Institute (ANSI), 2
ANSI C, 2
apice doppio \, 143
apice singolo \' , 143
argc (argument count), 314, 319
argomenti, 192, 201
argomenti costituiti da vettori, 277
argomenti della riga di comando, 314
argomenti delle macro vuoti, 343
argv (argument vector), 314, 319
aritmetica dei puntatori, 269, 283
assegnamento, 18, 20, 60
assegnamento composto, 62
assegnamento semplice, 60
atol, 709
auto, 477

B

backslash \\, 143
bitwise, 525-530
blocchi, 237
break, 91
build, 361, 378, 381

C

C89, C90, 2
calloc, 435, 466
campi etichetta, 412, 416
campo di minimo, 41
carattere null, 293, 317
caratteri signed e unsigned, 141
carriage return \r, 143, 159
case, 90
casting, 153
char, 140, 142
chiamata a funzione, 16, 197
classe di memorizzazione extern, 460, 478

classi di memorizzazione, 474, 475
clausola else, 80
codice di escape /t, 49
collegamento, 476
comma expression, 113
commenti, 17
compilatore GCC 13, 31, 94, 383
compilazione, 12, 328, 378
compilazione condizionale, 330, 332, 346, 349
comportamento definito dall'implementazione, 57
comportamento indefinito, 67
confrontare i puntatori, 272
const, 28, 178, 262, 266, 277, 494
conversione %i, 49
conversione degli argomenti, 202
conversione di tipo, 148
conversioni esplicite, 149
conversioni implicite, 149
conversioni implicite nel C99, 152
conversioni negli assegnamenti, 151
copiare un vettore, 182-183, 589
corpo (o body), 103, 192
costanti, 20, 25
costanti a virgola mobile esadecimali, 158
costanti di tipo carattere, 140
costanti floating point, 139, 158
costanti intere, 134

D

data pool, 502
definire e invocare le funzioni, 191
definizione di tipo (*type definition*), 155
definizione di una funzione, 195
designatore, 171, 392
designatore inizializzato, 171, 182
dichiaratori, 474, 483
dichiarazione, 19, 473
dichiarazione di funzioni, 199, 200, 219
dichiarazione di un parametro, 208, 284
dichiarazione extern, 28, 478
direttiva, 14
direttiva #elif, 348
direttiva #else, 348
direttiva #endif, 346

Ritchie, D. M., S.C. Johnson, M. E. Lesk e B.W. Kernighan, *Unix timesharing system: the C programming language* in *Bell System Technical Journal* 57, 6 (July-August 1978), 1991-2019. Un famoso articolo che discute delle origini del C e descrive il linguaggio così come si presentava nel 1978.

Rosler, L., *The UNIX system: the evolution of C – past and future* in *AT&T Bell Laboratories Technical Journal* 63, 8 (October 1984), 1685-1699. Le tracce dell'evoluzione del C dal 1978 al 1984 e oltre.

Summit, S., *C Programming FAQs: Frequently Asked Questions*, Addison-Wesley, Reading, Mass., 1996. Una versione espansa dell'elenco delle FAQ che è apparso per anni nel newsgroup Usenet *comp.lang.c*.

van der Linden, P., *Expert C Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1994. Scritta da uno dei maghi del C della Sun Microsystem, questo libro riesce in egual modo a intrattenere e a informare. Con la sua profusione di aneddoti e scherzi, fa sembrare un divertimento anche lo studio dei punti più sottili del C.

Programmazione UNIX

Rochkind, M.J., *Advanced UNIX Programming*, Second Edition, Addison-Wesley, Boston, Mass., 2004. Tratta le chiamate di sistema di UNIX con un dettagli considerevole. Questo libro, assieme a quello di Stevens e Rago, è un "must" per tutti i programmatore C che utilizzano il sistema operativo UNIX o una delle sue varianti.

Stevens, W.R. e S.A. Rago, *Advanced Programming in the UNIX Environment*, Second Edition, Addison Wesley, Upper Saddle River, N.J., 2005. Un eccellente libro per i programmatore che lavorano con il sistema operativo UNIX. Il libro si concentra sull'uso delle chiamate di sistema UNIX, includendo sia le funzioni della libreria standard del C che le funzioni che sono specifiche di UNIX.

Programmazione in generale

Bentley, J., *Programming Pearls*, Second Edition, Addison-Wesley, Reading, Mass., 2000. Questa versione aggiornata del classico libro di Bentley pone enfasi sulla scrittura di programmi efficienti, ma tocca anche altri argomenti che sono fondamentali per il programmatore professionista. Il tocco leggero dell'autore rende il libro sia divertente da leggere che informativo.

Kernighan, B.W. e R. Pike, *The Practice of Programming*, Addison-Wesley, Reading, Mass., 1999. Questo libro va letto per avere dei consigli sullo stile di programmazione, sulla scelta del giusto algoritmo, sul testing e sul debugging e sulla scrittura di programmi portabili. Gli esempi sono tratti dal C, dal C++ e da Java.

McConnel, S., *Code Complete*, Second Edition, Microsoft Press, Redmond, Wash., 2004. Cerca di riempire il vuoto esistente tra la teoria della programmazione e la pratica fornendo degli esempi realistici basati su ricerche dimostrate. Include tantissimi esempi

scritti in una varietà di linguaggi di programmazione diversi. È altamente raccomandato.

Raymond, E.S., a cura di, *The New Hacker's Dictionary*, Third Edition, MIT Press, Cambridge, Mass., 1996. Spiega molto del gergo utilizzato dai programmati oltre ad essere veramente divertente da leggere.

Risorse sul Web

ANSI eStandards Store (webstore.ansi.org). Lo standard C99 (ISO/IEC 9899:1999) può essere acquistato in questo sito. Ogni insieme di correzioni apportate allo standard (conosciute come Technical Corrigendum) possono essere scaricate gratuitamente.

comp.lang.c Frequently Asked Questions (c-faq.com). L'elenco di FAQ di Steve Summit per il newsgroup *comp.lang.c* deve essere assolutamente letto da ogni programmatore C.

Dinkumware (www.dinkumware.com). Dinkumware è di P.J. Plauger, il comprovato maestro delle librerie standard del C e del C++. Tra le altre cose, il sito web include una completa guida alla libreria C99.

Google Groups (groups.google.com). Uno dei modi migliori per trovare le risposte alle domande relative alla programmazione è quello di cercare nei newsgroup Usenet, utilizzando il motore di ricerca Google Groups. Se avete una qualsiasi domanda, è probabile che qualcun altro l'abbia già fatta su un newsgroup e che ci sia un post contenente la risposta. Gruppi di particolare interesse per i programmati C includono *alt.comp.lang.learn.c-c++* (per i principianti del C e del C++), *comp.lang.c* (il gruppo principale per il linguaggio C) e *comp.std.c* (dedicato alle discussioni sullo standard C).

International Obfuscated C Code Contest (www.ioccc.org). Sede della competizione internazionale nella quale i partecipanti si confrontano per vedere chi è in grado di scrivere i più oscuri programmi C.

ISO/IEC JTC1/SC22/WG14 (www.open-std.org/jtc1/sc22/wg14/). Il sito web ufficiale di WG14, il gruppo di lavoro internazionale che ha creato lo standard C99 e che è responsabile del suo aggiornamento. Di particolare interesse tra i molti documenti disponibili sul sito sono le ragioni del C99, che spiegano il perché delle modifiche fatte nello standard.

Lysator (www.lysator.liu.se/c/). Una collezione di link di siti web relativi al C a cura di Lysator, un'associazione accademica sul mondo dei computer presso la svedese Linköping University.

- d**
 direttiva #error, 350
 direttiva #if, 346
 direttiva #ifdef, 347
 direttiva #ifndef, 347
 direttiva #include, 363, 384
 direttiva #line, 351
 direttiva #pragma, 352
 direttiva nulla, 354
 directive del preprocessore, 240, 330
double _Complex type (C99), 743
double, floating point a doppia precisione, 138
 durata di memorizzazione, 475
 durata di memorizzazione automatica, 476
 durata di memorizzazione statica, 181, 230
E
 elenco di argomenti di lunghezza variabile, 464
 ellissi, 344, 704, 705
 else, 80
 enumerazioni, 88, 413
 errori durante il linking, 380
 espressione condizionale, 95
 espressioni, 64
 espressioni condizionali, 85
 espressioni logiche, 75
EXIT_FAILURE, 211-212
EXIT_SUCCESS, 211-212, 713
 expression statement, 67, 70, 197
- F**
fexcept_t type (C99), 759
file header, 363, 384
float _Complex type (C99), 743
float, floating point a singola precisione, 138
floating-point, flag di stato, 755
floating-point, modalità di controllo, 755
 form feed \f, 143
 free, 437
 freestanding implementation, 356
 funzioni, 14
 atol, 709
 auto, 477
 break, 91
 exit, 118, 211
 fgetpos, 592, 682
 fsetpos, 592, 682
 frexp, 616
 getchar, 146
 gets, 297
 getwc (C99), 685
 getwchar (C99), 685
 longjmp, 658
 malloc, 429, 430, 465
 memcmp, 639
- printf, 39, 296, 317
 putchar, 146
 puts, 296
 scanf, 45, 297, 317
 srand, 179
 strcat, 303
 strcmp, 304
 strcpy (string copy), 301
 strlen, 303
 strtol, 709
 time, 179
 toupper, 144
 vprintf, 706
 vscanf (C99), 707
 vsprintf, 706
 vsscanf (C99), 707
 funzioni estendibili di classificazione dei wide-character, 696
 funzioni estendibili per la mappatura dei wide-character, 697
 funzioni per la manipolazione dei caratteri, 144
 funzioni ricorsive, 213, 215, 247
- G**
 garbage, 437
 GCC, 13, 31, 94, 383
 gestione degli errori, 518, 649
 getchar, 146, 160
 gets, 297
 getwc (C99), 685
 getwchar (C99), 685
 goto, 115, 117, 183
- H**
 header <limits.h>, 133, 612
 hosted implementation, 356
- I**
 I macro (C99), 745
 I/O formattato, 569
 identificatori, 27
 if, 78, 82
 implementazione freestanding, 342
 implementazione hosted, 342
 encapsulamento, 508
 indentazione, 94
 indicizzazione del vettore, 283
 indicizzazione o subscripting del vettore, 168
 information hiding, 503
 inizializzare un vettore di strutture, 400
 inizializzatore, 23, 170, 487
 inizializzatore designato, 209, 392
 inizializzazione, 23

inizializzazione dei vettori, 170

inline, funzioni, 489-492

integer overflow, 136

integral types, 142

interi signed, 131

ISO (International Organization for Standardization), 2

ISO/IEC 9899:1990, 2

ISO/IEC 9899:1999, 2

istruzioni

break, 91, 115

composta, 79, 104

continue, 116

do, 107

for, 109

goto, 117

if in cascata, 82

if, 78

switch, 89

vuota, 121

while, 103

K

keyword, 28

keyword extern, 368, 478

keyword restrict, 460

L

label, 117

leggere e scrivere caratteri con scanf e printf, 145

leggere e scrivere caratteri usando le funzioni getchar e putchar, 146

letterale composto, 208, 398, 418, 492

libreria, 502

libreria C per le stringhe, 300

line-feed, 159

linking, 12, 378, 385

lint, 6, 8

lista concatenata, 438

lista ordinata, 447

long double, floating point con precisione estesa, 138

long double _Complex type (C99), 743

longjmp, 658

lvalue, 61, 69, 393

M

macro, 327

macro MB_CUR_MAX, 671

macro con un numero variabile di argomenti, 344

macro, definizione di, 25

macro NULL, 314

macro parametriche, 333

macro predefinite, 341

macro semplici, 331

main, 221

make, 385

makefile, 379

malloc, 430, 465

membri, 389

membri vettore flessibili, 462

memcmp, 639

memorizzazione automatica, durata della, 229

memorizzazione statica, durata della, 247

memorizzazione (o estensione) di una variabile, durata della, 229

memory leak, 437

moduli, 500-503

N

new-line \n, 143

nome di un vettore, 275

nomi universali per i caratteri (universal character names), 144

normali conversioni aritmetiche, 152

normalizzati, 138

NULL, 463, 464

numeri esadecimali, 49

numeri ottali, 49

O

oggetto astratto, 503

operatore associativo a destra, 58

operatore associativo a sinistra, 58

operatore asterisco, 256

operatore condizionale, 85

operatore defined, 347

operatore di assegnazione (=), 79

operatore di indicizzazione, 220

operatore di uguaglianza (==), 79

operatore indirizzo, 255

operatore sizeof, 157, 204, 284

operatore virgola, 113

operatori,

##, 354

#, 336, 354

& (indirizzo), 255

* (indirezione), 255, 265

*, 274

-, 274

_Pragma, 353

++, 274

->, 440

operatori aritmetici, 56

operatori bitwise, 525

operatori di assegnamento, 60

operatori di decremento, 63-64

operatori di incremento, 63-64

operatori di uguaglianza, 76
 operatori logici, 77
 operatori relazionali, 76
 operazioni sui caratteri, 141

P

parametro costituito da un vettore, 207, 220, 278
 portabilità, 2, 4, 156
 posizionamento nei file, 561, 590
 precedenza degli operatori e associatività, 57
 preprocessing, 12
 promozione di default degli argomenti, 201, 202, 366
 promozioni integrali, 149, 160, 203
 proprietà generali delle macro, 337
 prototipi di funzione, 201, 218, 229, 240
 puntatore a file, 556
 puntatore nullo, 314, 428
 puntatore restricted, 460
 puntatori a funzioni, 454
 puntatori a letterali composti, 272
 puntatori a puntatori, 452
 puntatori e vettori a lunghezza variabile (C99), 282
 punto di domanda ?, 143
 printf, 39, 296, 317
 putchar, 146
 puts, 296
 putwc (C99), 685
 putwchar (C99), 685

Q

quint, 455-457, 467
 quirksoft, 216, 468
 qualificatore const, 178
 qualificatori di tipo, 482

R

realloc, 446
 regalloc, 470
 rendimento, 557
 rendimento dell'input, 372, 557
 rendimento dell'output, 372, 557
 return, 12, 16, 32, 79, 209
 reservation, 212-213, 222
 rvalue, 89

sequenza di escape ottale, 144, 240
 sequenze trigrafiche (trigraph sequences), 144
 set di caratteri, 140
 ASCII, 140, 305
 Latin-1, 140
 setjmp macro, 658
 side effect, 61, 68, 69, 335
 sig_atomic_t type, 736
 sistemi di sviluppo integrati, 13
 size_t type, 157, 736
 sizeof, 173, 416
 spazio dei nomi, 391
 specifica di conversione %, 49
 specifica di conversione %p, 265
 specifiche %o e %x, 42, 157
 specifiche di conversione, 39, 40
 srand, 179
 stack o pila, 231
 standard ANSI X3.159-1989, 2
 standard floating point dell'IEEE, 138
 static, 208, 477
 strcat, 303
 strcmp, 304, 318
 strcpy (string copy), 301
 stream, 556
 stringa di formato, 24, 39, 43, 47
 stringhe letterali, 289, 316
 stringhe, 16, 289
 strlen, 303
 strtol, 709
 strutture annidate, 399
 strutture dati composite, 411
 strutture di tipo compatibile, 394
 switch, 89, 95

T

tag di struttura, 394, 466
 time, 179
 tipi aritmetici (arithmetic types), 142, 149, 153
 tipi complessi (float _Complex, double _Complex, long double _Complex), 143
 tipi di dato astratti, 507
 tipi floating point (float, double, long double), 138,
 142, 143
 tipi generici, invocare una macro per i, 753
 tipi incompleti, 508, 521
 tipi integrali (integral types), 142
 tipi interi, 131
 tipi enumerati, 142
 tipi estesi, 142
 tipi interi con segno (signed char, short int, int, long int),
 142
 tipi interi senza segno (unsigned char, unsigned short int,
 unsigned int, unsigned long int), 142
 tipi interi nel C99, 134

tipi unsigned, 131
tipo di dato astratto, 503
tipo incompleto, 463
token, 29, 29, 679
token ... (ellissi), 344, 704, 705
token del preprocessore, 331
typedef, 155, 417

U

universal character names, 27
unione, 408

V

valori booleani (C89), 87
valori booleani (C99), 88
variabili, 18
variabili esterne, 231, 232
variabili globali, 231
variabili locali, 229
variabili locali statiche, 230

variabili stringa, 292
variabili struttura, 389
vettore (array), 167
vettore a lunghezza variabile, 161, 181, 183, 494
vettore di puntatori, 314
vettori a lunghezza variabile usati come argomenti, 203
vettori allocati dinamicamente, 434
vettori costanti, 178
vettori di strutture, 399
vettori multidimensionali, 175, 279
vettori usati come argomenti, 203
VLA, 181
vprintf, 706
vscanf (C99), 707
vsprintf, 706
vsscanf (C99), 707

W

wchar_t type, 343, 686
wcstoumax (C99), 740
wint_t type (C99), 736