

# Il linguaggio C

con MyLab

## Libro digitale

- Versione del libro in HTML5

The screenshot shows the Pearson MyLab interface. At the top, there's a navigation bar with 'MyLab Italia', 'Area personale', 'KERNIGHAN ITY', and 'Altre classi'. Below this is a sidebar with 'Presentazioni del corso' (Il linguaggio C), 'Accedi a MyLab', 'Informazioni sulla classe', and 'Impostazioni'. The main content area displays the title 'Il linguaggio C' with a small thumbnail image. A descriptive text block follows, detailing the digital learning environment and its features like audio playback and annotation tools. At the bottom, there's a 'Dashboard' section with tabs for 'In programma', 'Preferiti', 'Nascosto', and 'Precedente', and a note stating 'Non sono presenti voci in programma.'

Per accedere alla piattaforma collegati al sito

[www.pearson.it/place](http://www.pearson.it/place) con:

- il tuo indirizzo e-mail
- il codice di accesso che trovi in copertina

€ 29,00



9788891908230

### Gli autori

**Brian Wilson Kernighan** è uno scienziato informatico canadese che ha lavorato presso i Bell Labs insieme ai creatori di Unix Ken Thompson e Dennis Ritchie e ha contribuito allo sviluppo di Unix. È anche coautore dei linguaggi di programmazione AWK e AMPL. Dal 2000 Brian Kernighan è professore presso il Dipartimento di Informatica dell'Università di Princeton.

**Dennis MacAlistair Ritchie** (9 settembre 1941 - 12 ottobre 2011) era uno scienziato informatico americano. Ha creato il linguaggio di programmazione C e, con il collega di vecchia data Ken Thompson, il sistema operativo Unix. Ritchie era a capo del dipartimento di ricerca software di sistema di Lucent Technologies quando si ritirò in 2007.

Pearson

9788891908230A

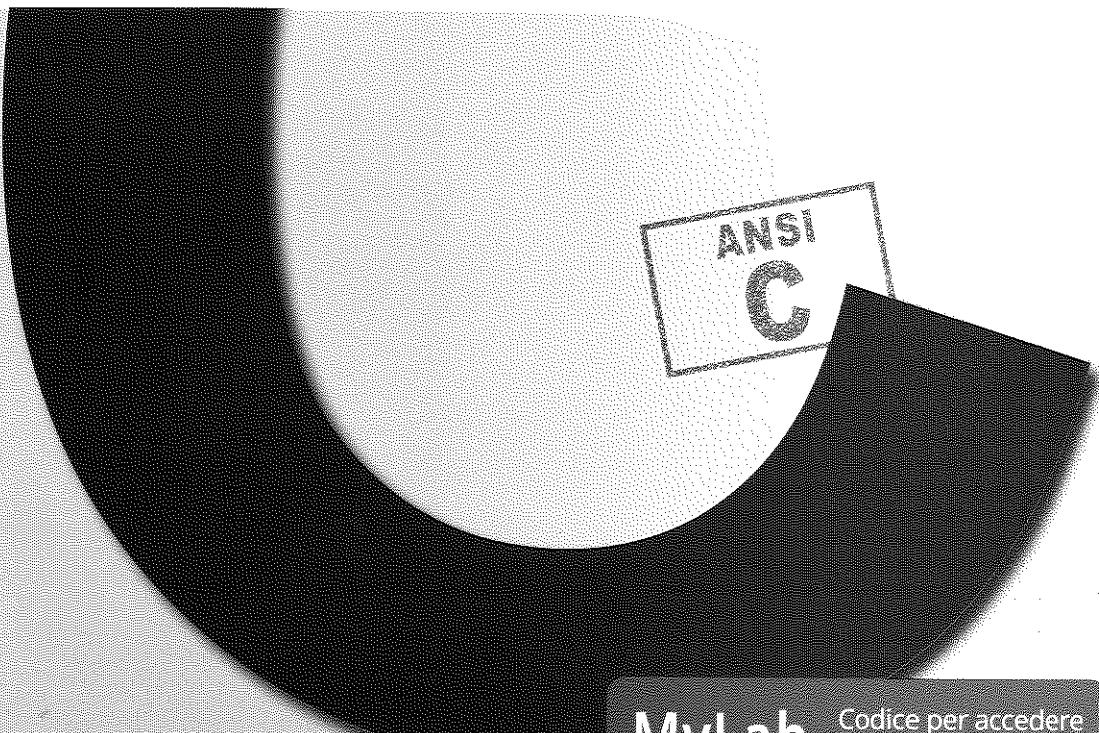
Brian W. Kernighan  
Dennis M. Ritchie

# Il linguaggio C

Principi di programmazione  
e manuale di riferimento

Seconda edizione

Brian W. Kernighan  
Dennis M. Ritchie



MyLab

Codice per accedere  
alla piattaforma

CODICE STUDENTE MONOUSO

ISBN 9788891908230B

Attivabile dal 05/09/17 fino al 31/12/27. Durata 18 mesi.

L R218035A

Pearson

Brian W. Kernighan  
Dennis M. Ritchie

# Il linguaggio C

**Principi di programmazione e manuale di riferimento**  
**Seconda edizione**



# Sommario

© 2007 Pearson Paravia Bruno Mondadori S.p.A.

*Authorized translation from the English language edition, entitled THE C PROGRAMMING LANGUAGE, 2<sup>nd</sup> Edition by KERNIGHAN, BRIAN W; RITCHIE, DENNIS M., published by Pearson Education, Inc, publishing as Prentice Hall PTR, Copyright © 1988*

*All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.*

*Italian language edition published by Pearson Paravia Bruno Mondadori S.p.A., Copyright © 2007.*

Le informazioni contenute in questo libro sono state verificate e documentate con la massima cura possibile. Nessuna responsabilità derivante dal loro utilizzo potrà venire imputata agli Autori, a Pearson Paravia Bruno Mondadori S.p.A. o a ogni persona e società coinvolta nella creazione, produzione e distribuzione di questo libro.

I diritti di riproduzione e di memorizzazione elettronica totale e parziale con qualsiasi mezzo, compresi i microfilm e le copie fotostatiche, sono riservati per tutti i paesi.

LA FOTOCOPIATURA DEI LIBRI È UN REATO. Le fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume dietro pagamento alla SIAE del compenso previsto dall'art. 68, commi 4 e 5, della legge 22 aprile 1941 n. 633.

Le riproduzioni effettuate per finalità di carattere professionale, economico o commerciale o comunque per uso diverso da quello personale possono essere effettuate a seguito di specifica autorizzazione rilasciata da AIDRO, corso di Porta Romana n. 108, 20122 Milano, e-mail [segreteria@aidro.org](mailto:segreteria@aidro.org) e sito web [www.aidro.org](http://www.aidro.org).

Traduzione: Valerio Marra

Revisione tecnica: Vincenzo Marra

Realizzazione editoriale: **shortcut**

Grafica di copertina: Gianni Gilardoni

Stampa: Tip.Le.Co. - San Bonico (PC)

Tutti i marchi citati nel testo sono di proprietà dei loro detentori.

978-88-7192-200-3

Printed in Italy

1<sup>a</sup> edizione: febbraio 2004

Ristampa Anno

06 07 08 09 10 09 10 11 12 13

## Prefazione

### Prefazione alla prima edizione

### Introduzione

<b>1 Panoramica del linguaggio</b>	<b>1</b>
1.1 Primi passi	2
1.2 Variabili ed espressioni aritmetiche	4
1.3 Costrutto for	10
1.4 Simboli di costanti	11
1.5 Lettura e scrittura di caratteri	12
1.6 Vettori	18
1.7 Funzioni	21
1.8 Argomenti: chiamata per valore	24
1.9 Vettori di caratteri	25
1.10 Variabili esterne e visibilità	27
<b>2 Tipi, operatori ed espressioni</b>	<b>33</b>
2.1 Nomi delle variabili	34
2.2 Tipi e dimensioni dei dati	34
2.3 Costanti	35
2.4 Dichiarazioni	38
2.5 Operatori aritmetici	39
2.6 Operatori relazionali e logici	39
2.7 Conversioni di tipo	40
2.8 Operatori di incremento e decremento	44
2.9 Operatori per la manipolazione dei bit	46
2.10 Operatori di assegnamento ed espressioni	48
2.11 Espressioni condizionali	49
2.12 Precedenze e ordine di valutazione	50
<b>3 Flusso del controllo</b>	<b>55</b>
3.1 Istruzioni e blocchi	53
3.2 If-else	54
3.3 Else-if	55
3.4 Switch	56
3.5 Cicli: while e for	58
3.6 Cicli: do-while	61
3.7 Break e continue	62
3.8 Goto ed etichette	63

<b>4. Funzioni e struttura dei programmi</b>	<b>65</b>
4.1 Fondamenti delle funzioni	66
4.2 Funzioni che restituiscono valori diversi da interi	69
4.3 Variabili esterne	71
4.4 Regole di visibilità	78
4.5 Intestazioni	79
4.6 Variabili static	81
4.7 Variabili register	81
4.8 Struttura a blocchi	82
4.9 Inizializzazione	83
4.10 Ricorsione	84
4.11 Il preprocessore del C	86
<b>5 Puntatori e vettori</b>	<b>91</b>
5.1 Puntatori e indirizzi	91
5.2 Puntatori e argomenti delle funzioni	93
5.3 Puntatori e vettori	96
5.4 Aritmetica degli indirizzi	99
5.5 Puntatori a caratteri e funzioni	102
5.6 Vettori di puntatori; puntatori a puntatori	105
5.7 Vettori multidimensionali	109
5.8 Inizializzazione dei vettori di puntatori	111
5.9 Puntatori e vettori multidimensionali a confronto	111
5.10 Argomenti dalla riga di comando	112
5.11 Puntatori a funzioni	117
5.12 Dichiarazioni complesse	121
<b>6 Strutture</b>	<b>127</b>
6.1 Fondamenti delle strutture	128
6.2 Strutture e funzioni	130
6.3 Vettori di strutture	132
6.4 Puntatori a strutture	137
6.5 Strutture autoreferenziali	139
6.6 Ricerca su tabelle	144
6.7 Typedef	146
6.8 Unioni	148
6.9 Campi di bit	149
<b>7 Input e output</b>	<b>153</b>
7.1 Standard input e output	153
7.2 Formattazione dei dati in uscita: printf	155
7.3 Liste di argomenti di lunghezza variabile	157
7.4 Formattazione dei dati in ingresso: scanf	159
7.5 Accesso ai file	162
7.6 Gestione degli errori: stderr ed exit	165
7.7 Input e output delle righe	167
7.8 Funzioni varie	168

<b>8 Interfaccia con il sistema UNIX</b>	<b>173</b>
8.1 Descrittori dei file	174
8.2 I/O a basso livello: read e write	174
8.3 Open, creat, close e unlink	176
8.4 Accesso casuale: lseek	178
8.5 Esempio di implementazione di fopen e getc	179
8.6 Esempio di liste di directory	184
8.7 Esempio di allocatore di memoria	190
<b>A Manuale di riferimento</b>	<b>197</b>
A.1 Introduzione	197
A.2 Convenzioni lessicali	197
A.3 Notazione della sintassi	201
A.4 Significato degli identificatori	201
A.5 Oggetti e lvalue	204
A.6 Conversioni	204
A.7 Espressioni	207
A.8 Dichiarazioni	219
A.9 Istruzioni	234
A.10 Dichiarazioni esterne	237
A.11 Campo di visibilità e linkage	240
A.12 Il preprocessore	242
A.13 Grammatica	248
<b>B Libreria standard</b>	<b>257</b>
B.1 Ingresso e uscita dei dati: <stdio.h>	258
B.2 Classi di caratteri: <cctype.h>	266
B.3 Funzioni per la gestione delle stringhe: <string.h>	267
B.4 Funzioni matematiche: <math.h>	268
B.5 Funzioni di utilità: <stdlib.h>	269
B.6 Diagnistica: <assert.h>	272
B.7 Liste di argomenti di lunghezza variabile: <stdarg.h>	272
B.8 Salti non locali: <setjmp.h>	272
B.9 Segnali: <signal.h>	273
B.10 Funzioni di data e ora: <time.h>	274
B.11 Limiti definiti dall'implementazione: <limits.h> e <float.h>	276
<b>C Sommario delle modifiche</b>	<b>279</b>
<b>Indice analitico</b>	<b>283</b>

# Prefazione

L MONDO DEI CALCOLATORI ha vissuto una rivoluzione da quando fu pubblicato, nel 1978, *The C Programming Language*. I grandi computer sono molto più grandi, e le capacità dei personal sfidano quelle dei mainframe di qualche decennio fa. Nel frattempo anche il C è cambiato, pur se in misura minore, diffondendosi ben al di là delle proprie origini come linguaggio del sistema operativo UNIX.

L'aumento di popolarità del C, le modifiche al linguaggio nel corso degli anni, e la creazione di compilatori da parte di gruppi estranei alla sua stesura, hanno contribuito a evidenziare il bisogno di una definizione del linguaggio più precisa e aggiornata di quella proposta dalla prima edizione. Nel 1983 l'American National Standards Institute (ANSI) ha fondato un comitato con l'obiettivo di produrre "una definizione del C chiara e indipendente dalla macchina", capace però di preservarne lo spirito originale. Il risultato è lo standard ANSI per il C.

Lo standard formalizza costrutti a cui si era accennato di sfuggita nella prima edizione, in particolare l'assegnamento fra strutture e le enumerazioni. Fornisce una nuova forma di dichiarazione delle funzioni che consente il controllo incrociato fra le definizioni e il loro uso. Specifica una libreria standard, con una vasta gamma di funzioni attinenti all'ingresso e all'uscita dei dati, la gestione della memoria, delle stringhe, e operazioni simili. Precisa il funzionamento di caratteristiche non ben delineate nella definizione originale, e allo stesso tempo stabilisce quali aspetti del linguaggio rimangono dipendenti dalla macchina.

La presente edizione descrive il C così come è definito dallo standard ANSI. Pur avendo messo in risalto gli sviluppi del linguaggio, abbiamo deciso di scrivere unicamente nella nuova forma. Ciò non comporta, il più delle volte, differenze apprezzabili; il cambiamento più evidente è la nuova forma delle dichiarazioni e definizioni delle funzioni. I compilatori moderni integrano già molte caratteristiche dello standard.

Abbiamo tentato di mantenere la brevità della prima edizione. Il C non è un linguaggio di grandi dimensioni, dunque sarebbe stato fuori luogo concepire un'opera ingombrante. È stata perfezionata la trattazione di caratteristiche preminenti, quali i puntatori, che sono di vitale importanza per la programmazione in C. Oltre a migliorare gli esempi originali, ne abbiamo aggiunti altri in diversi capitoli. Per esempio, la trattazione delle dichiarazioni complesse è stata ampliata con l'aggiunta di programmi che convertono le dichiarazioni in descrizioni verbali e viceversa. Come nella prima edizione, tutti gli esempi sono stati verificati direttamente dal testo: i brani di codice sono in forma leggibile dalla macchina.

L'Appendice A, il manuale di riferimento, non è lo standard, ma rappresenta il tentativo di esporre i fondamenti dello standard in uno spazio minore. Il manuale è inteso come strumento di agevole comprensione per i programmati, ma non come definizione del linguaggio per i progettisti di compilatori; tale ruolo compete propriamente allo standard stesso. L'Appendice B è un riepilogo delle funzionalità della libreria standard. L'appendice C riassume in sintesi le novità rispetto alla versione originale.

Come rimarcato nella Prefazione alla prima edizione, il C "invecchia bene con l'aumentare dell'esperienza che se ne fa". A un decennio di distanza, possiamo confermarlo. Ci auguriamo che questo libro aiuti il lettore a imparare il C e utilizzarlo proficuamente.

Siamo profondamente debitori verso gli amici che ci hanno aiutato a realizzare questa seconda edizione. Jon Bentley, Doug Gwyn, Doug McIlroy, Peter Nelson e Rob Pike ci hanno fornito commenti acuti su quasi tutte le pagine del manoscritto. Ringraziamo per l'attenta lettura Al Aho, Dennis Allison, Joe Campbell, G.R. Emelin, Karen Fortgang, Allen Holub, Andrew Hume, Dave Kristol, John Linderman, Dave Prosser, Gene Spafford e Chris Van Wyk. Utili suggerimenti ci sono venuti anche da Bill Cheswick, Mark Kernighan, Andy Koenig, Robin Lake, Tom London, Jim Reeds, Clovis Tondo e Peter Weinberger. Dave Prosser ha risposto in merito a molti dettagliati interrogativi sullo standard ANSI. Abbiamo fatto largo uso del traduttore C++ di Bjarne Stroustrup per verificare in loco i nostri programmi, mentre un compilatore ANSI C per le prove finali ci è stato fornito da Dave Kristol. Rich Drechsler è stato di grande aiuto per la cura tipografica del testo.

A tutti il nostro grazie più sincero.

Brian W. Kernighan  
Dennis M. Ritchie

## Prefazione alla prima edizione

IL C È UN LINGUAGGIO DI PROGRAMMAZIONE di uso e applicazione generale caratterizzato da economia di espressione, flusso del controllo e strutture dei dati aggiornati, nonché da una vasta gamma di operatori. Il C non è un linguaggio "grande", né "di livello molto alto", e non è specialistico per una particolare area di applicazione, ma in molte circostanze la sua generalità e la mancanza di restrizioni lo rendono più utile ed efficace di linguaggi apparentemente più potenti.

Il C fu originariamente progettato e scritto per il sistema operativo UNIX, sul DEC PDP-11, da Dennis Ritchie. Il sistema operativo, il compilatore C e in sostanza tutti i programmi applicativi di UNIX (inclusi quelli usati per la preparazione di questo libro) sono scritti in C. Esistono compilatori anche per varie altre macchine, compreso l'IBM System/370, l'Honeywell 6000 e l'Interdata 8/32. Il C non è tuttavia vincolato ad alcuna macchina o sistema particolare, ed è facile scrivere programmi direttamente eseguibili da qualsiasi elaboratore che supporti il C.

Quest'opera si propone di aiutare il lettore ad apprendere come si programma in C. Contiene un'introduzione che guida gli utenti a muovere i primi passi il più rapidamente possibile, capitoli dedicati alle caratteristiche più importanti, e un manuale di riferimento. Il metodo di lavoro adottato privilegia la lettura, la scrittura e le modifiche degli esempi, piuttosto che l'elencazione di regole pura e semplice. Gli esempi sono per la maggior parte dei programmi veri e completi, anziché frammenti isolati. Tutti gli esempi sono stati verifi-

cati direttamente dal testo, che è in forma leggibile dalla macchina. Oltre a mostrare in concreto come si usa il linguaggio, abbiamo anche tentato, ove possibile, di illustrare algoritmi utili, principi di eleganza stilistica e progettazione robusta.

L'opera non è un manuale introduttivo alla programmazione; presuppone infatti una certa dimestichezza con nozioni di base della programmazione quali variabili, istruzioni di assegnamento, cicli e funzioni. Ciononostante, un programmatore alle prime armi dovrebbe essere in grado di capire e assimilare il linguaggio attraverso la lettura del testo, anche se gli eventuali consigli di un collega più esperto saranno d'aiuto.

Per la nostra esperienza, il C si è rivelato un linguaggio piacevole, espressivo e versatile per una gran quantità di programmi. È di facile apprendimento, e invecchia bene con l'aumentare dell'esperienza che se ne fa. Il nostro auspicio è che questo libro aiuti il lettore a usarlo con profitto.

Le riflessioni e i consigli ponderati di molti amici e colleghi hanno apportato un grande contributo a questo libro e accresciuto la nostra soddisfazione nello scriverlo. In particolare, Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy, Bill Roome, Bob Rosin e Larry Rosler hanno letto scrupolosamente molteplici versioni. Siamo grati anche ad Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson e Peter Weinberger per i loro utili commenti in diverse fasi, e ringraziamo Mike Lesk e Joe Ossanna per l'inestimabile aiuto nella composizione tipografica.

Brian W. Kernighan  
Dennis M. Ritchie

## Introduzione

**I**L C È UN LINGUAGGIO DI PROGRAMMAZIONE di uso e portata generale. Lo si è associato strettamente al sistema UNIX, all'interno del quale venne sviluppato, in quanto sia il sistema che molti dei suoi programmi sono scritti in C. Il linguaggio, comunque, non è vincolato ad alcuna macchina o sistema operativo, e sebbene sia stato chiamato "linguaggio di programmazione di sistema" poiché si presta alla scrittura di compilatori e sistemi operativi, altrettanto efficacemente è stato adoperato per creare programmi di utilità primaria in ambiti diversi.

Molte idee basili del C derivano dal linguaggio BCPL, sviluppato da Martin Richards. L'influenza esercitata dal BCPL sul C è stata mediata dal linguaggio B, realizzato da Ken Thompson nel 1970 per il primo sistema UNIX sul DEC PDP-7.

BCPL e B sono linguaggi privi di tipi, o "non tipati", come si usa anche dire in gergo, mentre il C ne possiede una vasta gamma. I tipi fondamentali, detti anche "primitivi", sono i caratteri, i numeri interi e i numeri decimali di svariate dimensioni. Esiste poi una gerarchia di tipi derivati, che vengono formati mediante puntatori, vettori, strutture e unioni. Le espressioni si formano con operatori e operandi; qualunque espressione può essere un'istruzione, compresi assegnamenti o chiamate di funzione. I puntatori danno luogo a un'aritmetica degli indirizzi indipendente dalla macchina.

Il C fornisce i costrutti fondamentali per regolare il flusso del controllo, necessari per l'elaborazione di programmi ben strutturati: il raggruppamento delle istruzioni, l'attua-

ne di decisioni (`if-else`), la scelta di un caso fra quelli possibili (`switch`), l'uso di un ciclo con la condizione di arresto all'inizio (`while`, `for`) o alla fine (`do`), l'uscita anticipata da un ciclo (`break`).

I valori restituiti dalle funzioni possono essere sia di tipo primitivo sia di tipo derivato, come strutture, unioni o puntatori. Qualunque funzione può essere chiamata ricorsivamente. Le variabili locali sono tipicamente "automatiche", o create ex-novo a ogni chiamata. Le definizioni delle funzioni non possono essere annidate, ma le variabili sono dichiarabili in appositi blocchi. Le funzioni di uno stesso programma in C possono essere suddivise in file distinti, detti "file sorgente" (*source files*), da compilarsi separatamente. Le variabili possono essere interne a una funzione, esterne ma visibili solo all'interno di un determinato file sorgente, oppure visibili all'intero programma.

Esiste una fase preliminare alla compilazione, detta con un anglismo *preprocessing* (letteralmente, "pre-elaborazione"). Il suo scopo è quello di sostituire alle macro, che sono delle abbreviazioni convenzionali, il loro effettivo valore nel testo del programma, di includere altri file sorgente e di eseguire la compilazione condizionale.

Il C è considerato un linguaggio relativamente "di basso livello". Questa definizione non è peggiorativa; significa semplicemente che il C adopera lo stesso tipo di oggetti usati da molti elaboratori, ovvero caratteri, numeri e indirizzi, che possono essere combinati tramite gli operatori aritmetici e logici di cui si servono le macchine reali.

Il C non fornisce comandi che interagiscano direttamente con oggetti complessi come le stringhe di caratteri, gli insiemi, le liste o i vettori, né vi sono comandi che agiscano su intere stringhe o vettori, benché le strutture possano essere copiate come un oggetto unico. Il linguaggio non offre strumenti specifici per allocare memoria, eccetto le definizioni statiche e la pila locale delle funzioni. Non è previsto un meccanismo automatico di deallocazione della memoria inutilizzata: in gergo, non è presente la *garbage collection* (letteralmente, "raccolta dei rifiuti"). Di per sé il C non offre neppure funzionalità di input/output (cioè immissione e produzione dei dati), non ha operazioni come READ o WRITE, e nemmeno alcun metodo incorporato di accesso ai file. Tutti questi meccanismi di alto livello devono essere attuati con funzioni chiamate esplicitamente, e molte implementazioni del linguaggio prevedono una raccolta ragionevolmente omogenea di tali funzioni.

Nello stesso spirito, il C permette solo il controllo di un singolo flusso di computazione: condizioni, cicli, raggruppamenti e sottoprogrammi, ma non multiprogrammazione, operazioni parallele, sincronizzazione o co-routine.

Nonostante l'assenza di alcune caratteristiche possa apparire come una grave lacuna ("Devo chiamare una funzione per confrontare due stringhe di caratteri?"), la dimensione ridotta del linguaggio porta benefici apprezzabili. Per esempio, il C si può descrivere in breve e imparare rapidamente. Un programmatore può legittimamente aspettarsi di apprendere, capire, nonché usare regolarmente tutto il linguaggio.

Per molti anni, il "Manuale di riferimento" della prima edizione di questo libro (presentato nell'Appendice A) ha costituito la definizione del C. Nel 1983, l'Istituto Nazionale

Americano per gli Standard (ANSI, American National Standards Institute) fondò un comitato al fine di giungere a una definizione moderna ed esauriente del C. Verso la fine del 1988 lo standard di definizione che ne risultò fu il cosiddetto "ANSI C". Molte delle relative definizioni sono già state acquisite dai moderni compilatori.

Lo standard ANSI si basa sul manuale di riferimento originale: il linguaggio è stato cambiato relativamente di poco, poiché uno degli obiettivi del comitato era di assicurare che gran parte dei programmi esistenti rimanesse valida, o almeno che i compilatori potessero segnalare le innovazioni introdotte.

Per la maggioranza dei programmatore il cambiamento più rilevante consiste nella nuova sintassi da applicare alle funzioni: la dichiarazione di una funzione può ora comprendere una descrizione degli argomenti della funzione, e la sintassi della definizione si evolve di conseguenza. Queste informazioni aggiuntive facilitano i compilatori nell'individuazione degli errori causati da argomenti incoerenti con il tipo atteso dalla funzione; la nostra esperienza conferma che questa aggiunta al linguaggio è molto utile.

Il linguaggio ha subito altri cambiamenti di minore entità: l'assegnamento di strutture e le enumerazioni, che erano già ampiamente disponibili, sono diventate ufficialmente parte del linguaggio; le computazioni in virgola mobile (o, con termine inglese diffuso anche in italiano, *floating point*) si possono ora eseguire con precisione singola; le proprietà dell'aritmetica, soprattutto per i tipi privi di segno, sono state chiarite; il preprocessore è più elaborato. Molti di questi cambiamenti avranno un impatto trascurabile sui programmatori.

Il secondo contributo rilevante dello standard è la definizione di una libreria di funzioni a sostegno del C. Essa include funzioni di accesso al sistema operativo (per esempio, per leggere e scrivere file), di formattazione dei dati in ingresso e uscita, di allocazione di memoria, di manipolazione di stringhe, e così via. Una serie di intestazioni convenzionali, dette appunto *header*, consente accesso uniforme alle dichiarazioni di funzioni e dei tipi di dati. I programmi che usano questa libreria per interagire con il sistema sottostante offrono la garanzia di un comportamento compatibile. La maggior parte della libreria è stata creata sul modello della "libreria standard per I/O" ("I/O" abbrevia "Input e Output") del sistema UNIX, descritta nella prima edizione, e ampiamente usata anche da altri sistemi.

Poiché i tipi di dati e le strutture di controllo del C sono direttamente comprensibili alla maggioranza degli elaboratori, la realizzazione di programmi autonomi richiede una libreria per la fase di esecuzione (*run-time*) molto ridotta. Le funzioni della libreria standard si chiamano esplicitamente, ed è possibile quindi evitarle quando non servono. Molte possono essere scritte in C e, tranne per i dettagli del sistema operativo che nascondono, sono esse stesse portabili.

Sebbene il C sfrutti appieno le potenzialità di molti elaboratori, è indipendente da qualsiasi architettura di macchina particolare. Con un po' di attenzione è facile scrivere programmi portabili, vale a dire eseguibili da una vasta gamma di elaboratori senza bisogno di modifiche. Lo standard mette in risalto le questioni di portabilità, e prescrive una serie di costanti che caratterizzano la macchina su cui gira il programma.

Il C non è un linguaggio "fortemente tipato", nel senso che non adotta una politica di coerenza dei tipi dei dati assolutamente rigida; nella sua evoluzione, però, ha conosciuto

un rafforzamento del controllo sui tipi. La versione originale del C tollerava (controvoglia) l'interscambio tra puntatori e numeri interi, una caratteristica che è stata eliminata da tempo: lo standard richiede ora le dichiarazioni proprie e le conversioni esplicite, già introdotte dai compilatori più accorti. Le nuove dichiarazioni di funzione fanno un altro passo in questa direzione. I compilatori rileveranno la gran parte degli errori di tipo, e non è prevista la conversione automatica dei tipi di dati incompatibili. Malgrado ciò, il C preserva la sua filosofia originale, secondo cui i programmatore sanno quello che fanno, e richiede soltanto che le loro intenzioni siano espresse con chiarezza.

Il C, come ogni altro linguaggio, ha i propri difetti. Qualche operatore ha un livello di precedenza scorretto, e alcune parti della sintassi potrebbero essere migliori; tuttavia il C si è rivelato un linguaggio molto espressivo e di notevole efficacia per un'ampia gamma di applicazioni.

Il libro è strutturato nel modo seguente. Il Capitolo 1 tratta il nucleo essenziale del C, così da avviare velocemente il lettore all'uso del linguaggio. Infatti crediamo fermamente che il modo più efficace di apprendere un nuovo linguaggio sia utilizzarlo subito per scrivere programmi. Il capitolo postula una discreta conoscenza degli elementi di base della programmazione; non si sofferma dunque sugli elaboratori, sulla compilazione, o sul significato di un'espressione come  $n=n+1$ . Laddove possibile, abbiamo tentato di inserire esempi su tecniche di programmazione utili, ma il libro non vuol essere un'opera di riferimento su strutture di dati e algoritmi; se costretti a scegliere,abbiamo privilegiato il linguaggio.

I Capitoli dal 2 al 6 trattano e analizzano vari aspetti del C, con più rigore formale rispetto al primo, benché si insista a dare priorità ai programmi completi piuttosto che a singoli brani di istruzioni. Il Capitolo 2 affronta i tipi di dati elementari, gli operatori e le espressioni. Il Capitolo 3 tratta del flusso del controllo: `if-else`, `switch`, `while`, `for` e così via. Il Capitolo 4 si occupa delle variabili e della struttura del programma (variabili esterne, regole di visibilità, file sorgente multipli, e via dicendo), e accenna anche al preprocessore. Il Capitolo 5 esamina i puntatori e l'aritmetica degli indirizzi. Il Capitolo 6 concerne le strutture e le unioni.

Il Capitolo 7 descrive la libreria standard, che fornisce un'interfaccia uniforme al sistema operativo. La libreria, definita dallo standard ANSI, è concepita per funzionare su tutte le macchine che supportano il linguaggio C: i programmi che la usano per ottenere accesso alle funzionalità del sistema operativo saranno quindi trasferibili da un sistema all'altro senza variazioni.

Il Capitolo 8 descrive un'interfaccia tra i programmi in C e il sistema operativo UNIX, ed è incentrato sull'input/output, sul file system e sull'allocazione di memoria. Sebbene parte del capitolo sia specificamente dedicata ai sistemi UNIX, anche i programmatore che usano altri sistemi vi troveranno materiale utile, compreso un approfondimento su come sia realizzata una versione della libreria standard e suggerimenti sulla portabilità.

L'Appendice A contiene un manuale di riferimento del linguaggio. Il testo ufficiale per la sintassi e la semantica del C è costituito proprio dallo standard ANSI. Quel documento, però, si rivolge principalmente agli autori di compilatori. Il manuale di riferimento conte-

nuto in questa appendice propone una definizione del linguaggio più sintetica e, sul piano dello stile, meno formale. L'Appendice B è un sommario della libreria standard, anch'esso adatto più a chi usa i programmi che a chi li scrive. L'Appendice C raccoglie in breve i cambiamenti apportati al linguaggio originale. In casi dubbi, comunque, lo standard e il proprio compilatore rimangono le autorità definitive sul linguaggio.

### **Nota dell'Editore per la nuova edizione italiana**

Il libro di Kernighan e Ritchie è un classico universalmente noto e apprezzato, su cui si sono formate intere generazioni di studenti e programmatore. Questa nuova edizione italiana – basata sulla seconda edizione originale, quella definitiva, aggiornata allo standard ANSI – restituisce finalmente un testo fondamentale, che non poteva mancare dal panorama editoriale italiano.

Rispetto alla precedente edizione, realizzata da altri e dal 2003 non più disponibile, questa pubblicazione presenta interessanti novità. Vale la pena sottolineare alcune scelte strategiche che rappresentano un ulteriore valore aggiunto per questa edizione quale, per esempio, quella di integrare direttamente, nel testo o in nota, l'*errata corrigere* presente sul sito <http://cm.bell-labs.com/cm/cs/cbook/>, costantemente aggiornato e integrato dalle continue e precise revisioni da parte degli stessi Autori e dai suggerimenti dei tanti lettori.

Inoltre un ringraziamento va a Giulia Maselli, Donatella Pepe e Paolo Postinghel, il team di copy-editing e impaginazione che con tanta dedizione e professionalità ha lavorato al progetto, e a Vincenzo Marra, il revisore tecnico, sia per la meticolosa cura con cui ha impreziosito dal punto di vista lessicale e culturale la traduzione sia per l'impegno profuso nell'attento controllo degli aspetti contenutistici, mantenendo sempre una fattiva collaborazione con lo stesso Kernighan. Ringraziamo, ovviamente, anche Brian Kernighan per il suo attento e minuzioso controllo finale sulle bozze e sul codice, mirato a fornire ai lettori un testo non solo esaustivo ma anche contraddistinto in ogni dettaglio da un elevato livello qualitativo.

In conclusione, siamo lieti e orgogliosi di poter restituire ai lettori italiani questo libro di culto, con la speranza che il nostro sforzo possa essere apprezzato: fateci sapere.

Pearson Education Italia  
febbraio 2004

# Panoramica del linguaggio

**C**OMINCIAMO CON UNA RAPIDA INTRODUZIONE AL C. Il nostro intento è quello di mostrare gli elementi essenziali del linguaggio per mezzo di programmi completi, evitando però di impantanarci in troppi dettagli, regole ed eccezioni. Al momento, sacrificiamo le esigenze di completezza e precisione (fatti salvi gli esempi, che devono essere corretti), per mettere il lettore in condizione di scrivere programmi utili nel minor tempo possibile. In quest'ottica è indispensabile partire dagli elementi fondamentali: variabili e costanti, aritmetica, flusso del controllo, funzioni, e le prime nozioni di input e output. In questo capitolo tralascieremo invece gli elementi del C importanti per scrivere programmi più corposi, come i puntatori, le strutture, molti dei numerosi operatori del C, diversi costrutti per il flusso del controllo e la libreria standard.

Questa impostazione ha i suoi inconvenienti, il più evidente dei quali è che il resoconto completo dei singoli elementi del linguaggio è rimandato, sicché il capitolo, a causa della sua brevità, può risultare fuorviante. Inoltre, dato che il potenziale del C non è sfruttato appieno, gli esempi non sono incisivi ed eleganti come potrebbero. Si è cercato di ridurre al minimo questi effetti, ma è bene tenerne conto. Un altro inconveniente riguarda i capitoli successivi, che ripeteranno inevitabilmente parti di questo capitolo. Ci auguriamo che la ripetizione possa giovare più che annoiare.

Ad ogni modo, il programmatore esperto dovrebbe essere capace di selezionare il materiale di questo capitolo più adatto alle proprie esigenze. Il principiante dovrebbe invece in-

tegrarlo provando a scrivere da sé qualche piccolo programma simile a quelli del testo. Entrambi possono usarlo come una cornice teorica a cui agganciare le descrizioni più approfondate a partire dal Capitolo 2.

## 1.1 Primi passi

Scrivere dei programmi è l'unico modo per impadronirsi di un nuovo linguaggio di programmazione. Il primo programma da scrivere è lo stesso per tutti i linguaggi:

*Si scriva un programma che visualizzi la frase:*

ciao, mondo

Ecco il grosso ostacolo; per superarlo occorre essere in grado di redigere il testo del programma, compilarlo con successo, caricarlo, eseguirlo e scoprire dov'è finito il risultato. Una volta assimilati questi meccanismi, tutto il resto risulta relativamente facile.

Il programma per scrivere "ciao, mondo" è questo:

```
#include <stdio.h>

main()
{
    printf("ciao, mondo\n");
}
```

Il modo per eseguire il programma dipende dal sistema che si usa. Per fare un esempio, sul sistema operativo UNIX occorre redigere il programma in un file il cui nome termini con ".c", diciamo *ciao.c*, e poi compilarlo con il comando

cc *ciao.c*

Se non si sono commessi errori sintattici, come l'omissione di un carattere in un'istruzione, il compilatore – invocato dal comando "cc" – genererà tranquillamente un file eseguibile denominato *a.out*. Se si richiama *a.out* digitando:

*a.out*

esso visualizzerà:

ciao, mondo

Per altri sistemi le regole cambieranno: chiedete a un esperto.

Passiamo ora a spiegare alcune cose su questo programma. Un programma in C di qualunque grandezza consiste di *funzioni* e *variabili*. Una funzione contiene *istruzioni* che specificano quali operazioni devono essere effettuate, mentre le variabili memorizzano i valori usati durante l'esecuzione. Le funzioni del C sono analoghe alle funzioni del Fortran, o alle procedure e alle funzioni del Pascal. L'esempio precedente è una funzione chiamata *main* (letteralmente, "principale", "primaria"). Di norma si è liberi di assegnare alle funzioni il nome che più agrada, ma il caso di "*main*" è particolare, perché il vostro programma comincerà l'esecuzione all'inizio di *main*. Questo implica che ogni programma deve contenere una funzione *main*.

```
#include <stdio.h>
main()
{
    printf("ciao, mondo\n");
}
```

include informazioni sulla libreria standard  
definisce una funzione chiamata *main* che non si aspetta alcun argomento dalla funzione chiamante  
le istruzioni di *main* sono racchiuse in parentesi graffe  
*main* chiama la funzione *printf* della libreria per visualizzare la sequenza di caratteri "ciao, mondo", \n indica il carattere newline

Il primo programma in C

Per svolgere il suo lavoro, *main* chiamerà altre funzioni in aiuto, alcune scritte dal programmatore, altre presenti nelle librerie cui si ha accesso. La prima riga del programma,

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

dice al compilatore di includere le informazioni sulla libreria standard per l'input/output: questa istruzione appare all'inizio di molti file sorgente in C. La descrizione della libreria standard si trova nel Capitolo 7 e nell'Appendice B.

Uno dei metodi che consentono la comunicazione dei dati tra funzioni prevede che la funzione chiamante fornisca alla funzione chiamata una lista di valori, detti *argomenti*. Le parentesi tonde, poste dopo il nome della funzione, racchiudono la lista degli argomenti. Nel nostro esempio, la funzione *main* non si aspetta argomenti da parte di un'altra funzione chiamante; ecco perché la lista è vuota: () .

Le istruzioni che costituiscono una funzione sono racchiuse tra parentesi graffe. In questo caso la funzione *main* contiene una sola istruzione,

```
printf("ciao, mondo\n");
```

Una funzione è chiamata tramite il suo nome, seguito da un elenco degli argomenti racchiuso fra parentesi tonde: in questo caso, quindi, si chiama la funzione *printf* con l'argomento "ciao, mondo\n". Si tratta di una funzione della libreria che produce – si dice anche genericamente "stampa" o "scrive", da "print" – dei dati in uscita, in questo caso la frase tra virgolette. Per la grande maggioranza dei sistemi, l'istruzione ha l'effetto di visualizzare "ciao, mondo" sullo schermo dell'elaboratore.

Una successione di caratteri è comunemente detta *stringa di caratteri*. In modo più formale, si parla anche di "costanti (di tipo) stringa". In C, le stringhe di caratteri sono racchiuse fra virgolette, come in "ciao, mondo\n". Al momento useremo le stringhe esclusivamente come argomenti per *printf* e altre funzioni.

La sequenza \n che compare nella stringa è una notazione del C per il carattere *newline*, che ha l'effetto di spostare in avanti di una riga i dati in uscita, allineandoli al margine sinistro. Se si omette \n (un esperimento che vale la pena di fare), si scopre che l'avanzamento del cursore alla riga seguente non avviene.

È necessario adoperare `\n` per andare a capo: tentativi come il seguente

```
printf ("ciao, mondo
");
```

inducono il compilatore del C a generare un messaggio di errore.

La funzione `printf` non fornisce mai il carattere newline automaticamente, il che rende possibili diverse chiamate per la costruzione in più fasi di un'unica riga di output. Avremo potuto scrivere il nostro primo programma anche così:

```
#include <stdio.h>

main()
{
    printf("ciao, ");
    printf("mondo");
    printf("\n");
}
```

ottenendo un identico risultato.

Si noti che `\n` rappresenta un solo carattere. In C, una “sequenza di controllo” (*escape sequence*, letteralmente “sequenza di fuga”) come `\n` fornisce un meccanismo generale per rappresentare caratteri non visualizzabili o di difficile digitazione: per esempio, `\t` sta per tabulazione, `\b` per backspace (“cancellazione dell’ultimo carattere”), `\\"` per le virgolette e `\\\` per la barra inversa (*backslash*). L’elenco completo è contenuto nel Paragrafo 2.3.

**Esercizio 1.1** Si esegua il programma “ciao, mondo” sul proprio sistema. Come esperimento, si scriva il programma incompleto, per osservare quali segnalazioni di errore si ottengono.

**Esercizio 1.2** Si provi a scoprire cosa succede quando l’argomento di `printf` contiene `\c`, scegliendo per `c` un carattere che non dia luogo a una delle sequenze di controllo già illustrate.

## 1.2 Variabili ed espressioni aritmetiche

Il prossimo programma usa la formula  ${}^{\circ}C = (5/9)({}^{\circ}F - 32)$  per visualizzare la seguente tabella di temperature Fahrenheit e i loro equivalenti in gradi centigradi, ovvero Celsius.

0	-17
20	-6
40	4
60	15
80	26
100	37
120	48
140	60
160	71
180	82

200	93
220	104
240	115
260	126
280	137
300	148

Il programma è ancora formato dalla definizione di una funzione singola chiamata `main`. È più lungo di quello che ha prodotto “ciao, mondo”, ma non complicato. Introduce nuovi spunti, come i commenti, le dichiarazioni, le variabili, le espressioni aritmetiche, i cicli e la formattazione dei dati in uscita.

```
#include <stdio.h>

/* visualizza la tabella Fahrenheit-Celsius
   per fahr = 0, 20, ..., 300 */
main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;           /* valore minimo in gradi F nella tabella delle
                           temperature */
    upper = 300;         /* valore massimo in gradi F */
    step = 20;           /* intervallo fra due temperature in gradi F
                           adiacenti */

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Le due righe

```
/* visualizza la tabella Fahrenheit-Celsius
   per fahr = 0, 20, ..., 300 */
```

sono un *commento*, che in questo caso spiega brevemente cosa fa il programma. Il compilatore ignora tutti i caratteri compresi tra `/*` e `*/`: si può liberamente usare questo spazio per stilare un programma di più facile comprensione. I commenti possono essere inseriti ovunque siano ammessi spazi bianchi, tabulazioni e caratteri *newline*.

Nel C tutte le variabili devono essere dichiarate prima di essere usate, normalmente all’inizio della funzione, prima delle istruzioni eseguibili. Una *dichiarazione* rende esplicite le proprietà delle variabili, e si compone di un nome di tipo seguito da un elenco di variabili, per esempio:

```
int fahr, celsius;
int lower, upper, step;
```

Il tipo `int` indica che le variabili elencate rappresentano numeri interi, in contrasto con `float`, che sta per *floating point* (“virgola mobile” in gergo informatico), e indica la possibilità che il numero denotato dalla variabile contenga dei decimali, o in altre parole non sia un intero. L’intervallo di valori ammessi tanto per le variabili di tipo `int` che per quelle di tipo `float` dipende dalla macchina che si usa; sono molto diffusi i valori interi lunghi 16 bit, che vanno cioè da -32768 a +32767, come anche gli interi lunghi 32 bit. Un numero di tipo `float` è tipicamente una quantità di 32 bit, con almeno 6 cifre significative e una magnitudine compresa tra  $10^{-38}$  e  $10^{+38}$ .

Il C fornisce svariati tipi di dati fondamentali oltre a `int` e `float`, tra cui:

<code>char</code>	carattere - un singolo byte
<code>short</code>	intero piccolo
<code>long</code>	intero grande
<code>double</code>	numero a virgola mobile con doppia precisione

Anche le dimensioni di questi oggetti dipendono dalla macchina. Esistono anche *vettori*, *strutture* e *unioni* di questi tipi fondamentali, *puntatori* che li denotano e *funzioni* che li restituiscono; ne ripareremo a tempo debito.

La computazione nel programma di conversione delle temperature inizia con i *costrutti di assegnamento*:

```
lower = 0;
upper = 300;
step = 20;
fahr = lower;
```

che impostano le variabili ai loro valori iniziali. Le istruzioni singole terminano con un punto e virgola.

Dato che ogni riga della tabella è calcolata nello stesso modo, faremo uso di un ciclo che ripete lo stesso brano di programma per ogni riga della tabella. Il ciclo `while` serve a questo:

```
while (fahr <= upper) {
    ...
}
```

Ecco come funziona il ciclo. Prima di tutto viene esaminata la condizione tra parentesi tonde. Se è vera (ossia, se il valore di `fahr` è minore o uguale al valore di `upper`), il contenuto del ciclo è eseguito. Per contenuto del ciclo si intendono le istruzioni racchiuse tra parentesi graffe, dette il “corpo del ciclo” (in inglese, *body*). La condizione è poi riesaminata e, a patto che sia vera, il corpo è di nuovo eseguito. Quando la condizione risulta falsa (ossia, quando `fahr` diventa maggiore di `upper`) il ciclo si arresta, e l’esecuzione del programma continua con l’istruzione successiva, da intendersi come la prima istruzione che segue la graffa di chiusura del corpo. In questo caso, non essendovi altre istruzioni, il programma termina. Il corpo di `while` può consistere di diverse istruzioni racchiuse fra parentesi graffe, come nel convertitore di temperatura, o di una singola istruzione, nel qual caso le graffe si possono anche omettere, come in

```
while (i < j)
    i = 2 * i;
```

In entrambi i casi, faremo sempre rientrare con una tabulazione il corpo del ciclo, così da rendere evidenti le istruzioni controllate da `while` (un rientro singolo è qui rappresentato da quattro spazi). I rientri sottolineano la struttura logica del programma, e anche se i compilatori C non si curano dell’aspetto tipografico dei programmi, gli spazi e i rientri sono cruciali per agevolare i lettori. Consigliamo di scrivere una sola istruzione per riga e di lasciare spazi bianchi attorno agli operatori per rendere chiari i raggruppamenti. Ci sembra di minore importanza la posizione delle parentesi, anche se c’è chi è pronto a sostenere il contrario. Abbiamo adottato uno stile piuttosto diffuso, fra i tanti possibili; scegliete quello che preferite e usatelo con coerenza.

La maggior parte del lavoro avviene nel corpo del ciclo. La temperatura Celsius è calcolata e assegnata alla variabile `celsius` dall’istruzione

```
celsius = 5 * (fahr-32) / 9;
```

Invece di moltiplicare semplicemente per  $5/9$ , abbiamo moltiplicato per 5 e poi diviso per 9, perché in C, come in molti altri linguaggi, la divisione tra interi viene *troncata*, e la parte frazionaria viene eliminata. Dato che 5 e 9 sono di tipo intero,  $5/9$  sarebbe troncata a zero, ragion per cui tutte le temperature Celsius sarebbero riportate come zero.

Da questo esempio si capisce anche qualcosa in più sul funzionamento di `printf`. Si tratta di una funzione di uso generale per la formattazione dei dati in uscita, analizzata a fondo nel Capitolo 7. Il suo primo argomento è una stringa di caratteri da visualizzare, con ogni % che indica il punto in cui ogni argomento successivo (il secondo, il terzo e così via) deve essere sostituito, e in quale forma deve essere visualizzato. Per esempio, `%d` (*decimal integer*, cioè “intero in notazione decimale”) specifica un argomento intero, per cui l’istruzione

```
printf ("%d\t%d\n", fahr, celsius);
```

dà luogo alla stampa dei valori delle due variabili di tipo intero `fahr` e `celsius`, separati da una tabulazione (\t).

La prima espressione della forma % nel primo argomento di `printf` è associata al secondo argomento, la seconda espressione al terzo argomento, e così via; ci deve essere piena corrispondenza di numero e tipo, pena la restituzione di risposte sbagliate.

È bene segnalare che `printf` non fa parte del linguaggio C; non c’è input né output che sia definito dal C. In effetti, `printf` è solo una delle utili funzioni disponibili nella libreria standard a cui possono generalmente accedere i programmi in C. Il suo comportamento, comunque, è definito dallo standard ANSI, quindi le sue proprietà dovrebbero restare immutate con tutti i compilatori e le librerie conformi allo standard.

Al fine di concentrare l’attenzione sul C vero e proprio, non baderemo molto a input e output fino al Capitolo 7; solo allora prenderemo in esame la formattazione dei dati in uscita, cioè dell’output. Se si devono immettere numeri, si legga la trattazione della funzione `scanf` del Paragrafo 7.4. Si tratta di una funzione analoga a `printf` che però, invece di produrre dati, li acquisisce; informalmente, `scanf` “legge l’input”.

Il programma di conversione della temperatura solleva qualche problema. Il primo è che l’output non ha un bell’aspetto, perché i numeri non sono allineati a destra. La soluzione è semplice: se si specifica un’ampiezza dopo ogni `%d` nell’istruzione `printf`, i numeri saranno allineati a destra nelle colonne.

Per esempio, potremmo scrivere

```
printf("%3d %6d\n", fahr, celsius);
```

per mostrare il primo numero di ogni riga in un campo a tre cifre, e il secondo in un campo a sei cifre, in questo modo:

0	-17
20	-6
40	4
60	15
80	26
100	37
...	

Il problema più difficile, invece, è che avendo usato l'aritmetica dei numeri interi, le temperature Celsius non sono molto precise; per esempio, 0°F corrisponde in realtà a -17.8°C circa, e non a -17°C. Per avere risposte più precise dovremmo usare numeri con virgola mobile invece degli interi, e questo ci impone qualche modifica nel programma. Eccone una seconda versione:

```
#include <stdio.h>

/* visualizza la tabella Fahrenheit-Celsius
   per fahr = 0, 20, ..., 300; versione con virgola mobile */
main()
{
    float fahr, celsius;
    int lower, upper, step;

    lower = 0;      /* valore minimo in gradi F nella tabella delle
                    temperature */
    upper = 300;    /* valore massimo in gradi F */
    step = 20;      /* intervallo fra due temperature in gradi F
                    adiacenti */

    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Questo programma è quasi uguale al precedente, eccetto che si usa `float` per definire `fahr` e `celsius` e la formula per la conversione è scritta in maniera più naturale. Non si sarebbe potuto usare la frazione 5/9 nella precedente versione, perché la divisione tra interi avrebbe troncato il risultato a zero. Tuttavia, il punto in una costante numerica, per esempio `5.0`, indica che essa è a virgola mobile, per cui `5.0/9.0` non viene troncato perché è il quoziente di due numeri non necessariamente interi. Adotteremo sempre la convenzione anglosassone di denotare l'inizio della parte frazionaria di un numero con il punto, anziché con la virgola com'è d'uso in Italia, per conformarci alla notazione usata da tutti i calcolatori.

Se un operatore aritmetico ha operandi interi, l'operazione effettuata è intera, mentre quando i due operandi sono rispettivamente a virgola mobile e intero, quello intero sarà convertito in virgola mobile prima di eseguire l'operazione. Se avessimo scritto `fahr-32`, il 32 sarebbe stato automaticamente convertito in un numero a virgola mobile. È indubbio comunque che indicare esplicitamente il separatore punto nelle costanti a virgola mobile anche quando esse abbiano parte frazionaria nulla ne evidenzia, dal punto di vista dei lettori, la reale natura. Le regole e i particolari per la conversione di interi in numeri a virgola mobile si trovano nel Capitolo 2. Intanto, si noti che l'assegnamento

```
fahr = lower;
```

e la condizione

```
while (fahr <= upper)
```

funzionano anch'essi in modo naturale: il valore di tipo `int` diventa `float` prima che l'operazione abbia luogo.

La specifica di conversione `%3.0f` in `printf` prescrive che il numero a virgola mobile in questione (qui è `fahr`) debba avere almeno tre caratteri di ampiezza, e che sia scritto senza punto decimale né parte frazionaria. La specifica `%6.1f` si riferisce a un altro numero (`celsius`) di almeno 6 caratteri di ampiezza, con una cifra dopo il punto decimale. Questo è il risultato che si ottiene:

0	-17.8
20	6.7
40	4.4
...	

L'ampiezza e la precisione possono essere trascurate: `%6f` colloca il numero in un campo di almeno sei caratteri, mentre `.2f` prescrive due caratteri dopo il punto decimale, ma permette all'ampiezza di variare, e `%f` significa semplicemente che il valore deve essere visualizzato come numero a virgola mobile.

<code>%d</code>	visualizza come intero in notazione decimale
<code>%6d</code>	visualizza come intero in notazione decimale, in un campo lungo almeno 6 caratteri
<code>%f</code>	visualizza come numero a virgola mobile
<code>%6f</code>	visualizza come numero a virgola mobile, in un campo lungo almeno 6 caratteri
<code>.2f</code>	visualizza come numero a virgola mobile, con 2 caratteri dopo il punto decimale
<code>%6.2f</code>	visualizza come numero a virgola mobile, in un campo lungo almeno 6 caratteri e con 2 caratteri dopo il punto decimale

Alcune altre espressioni utilizzabili da `printf` sono `%o` e `%x` (dall'inglese *octal* e *hexadecimal*), rispettivamente per la visualizzazione di interi in base otto e sedici, `%c` per i caratteri, `%s` per le stringhe di caratteri, e `%%` per lo stesso segno `%`.

**Esercizio 1.3** Si modifichi il programma di conversione delle temperature in modo da visualizzare un'intestazione sopra la tabella.

**Esercizio 1.4** Si scriva un programma che visualizzi una tabella di conversione delle temperature da Celsius a Fahrenheit.

### 1.3 Costrutto for

Esistono svariate possibilità di scrivere un programma che esegua un dato compito.

Proviamo ad applicare una variante al convertitore di temperature.

```
#include <stdio.h>

/* visualizza la tabella Fahrenheit-Celsius */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Il programma ha assunto un aspetto decisamente diverso, anche se genera gli stessi risultati. Un cambiamento notevole è l'eliminazione di tutte le variabili: rimane solo `fahr`, che abbiamo dichiarato come `int`. I limiti inferiore e superiore e la dimensione del passo sono presenti solo come costanti nell'istruzione `for`, che è un costrutto nuovo, e l'espressione che calcola la temperatura Celsius è ora all'interno del terzo argomento di `printf` invece di essere un'istruzione di assegnamento a sé stante.

Quest'ultima modifica costituisce l'applicazione di una regola generale: in ogni contesto dove sia consentito l'uso di una variabile di un certo tipo è possibile usare un'espressione più complessa dello stesso tipo. Dal momento che il terzo argomento di `printf` deve essere un numero a virgola mobile per corrispondere correttamente a `%6.1f`, qualsiasi espressione decimale è ugualmente ammessa.

L'istruzione `for` è un'iterazione o ciclo, come `while`, e ne costituisce una generalizzazione. Se la si confronta con il nostro precedente ciclo `while`, il suo modo di operare dovrebbe essere chiaro. Dentro le parentesi tonde vi sono tre parti, separate da un punto e virgola. La prima parte, eseguita una volta sola all'inizio dell'iterazione, è l'impostazione iniziale del valore ed è chiamata *inizializzazione*:

```
fahr = 0
```

La seconda parte è la condizione che controlla l'iterazione, o condizione per la terminazione del ciclo:

```
fahr <= 300
```

La condizione è valutata; se risulta vera, è eseguito il corpo del ciclo (in questo caso la sola istruzione `printf`). Quindi si passa all'esecuzione dell'incremento dell'indice del ciclo

```
fahr = fahr + 20
```

e la condizione viene di nuovo valutata. Il ciclo ha termine se la condizione diventa falsa. Come per `while`, il ciclo può avere come contenuto una singola istruzione, o una serie di istruzioni delimitate da parentesi graffe. L'inizializzazione, la condizione e l'incremento possono assumere la forma di qualunque espressione corretta.

La scelta tra `while` e `for` è arbitraria, guidata solo da considerazioni di opportunità quali la chiarezza del codice risultante. L'istruzione `for` si adatta meglio a cicli in cui l'inizializzazione e l'aumento siano istruzioni singole e logicamente correlate, dato che è più compatto di `while` e colloca tutte le istruzioni di controllo del ciclo in una singola posizione.

**Esercizio 1.5** Si modifichi il programma di conversione della temperatura per visualizzare la tabella in ordine inverso, cioè da 300 gradi a zero.

### 1.4 Simboli di costanti

Un'ultima osservazione prima di abbandonare la conversione di temperature: è sconsigliabile servirsi di "numeri magici" come i nostri 300 e 20 in un programma, perché forniscano un'informazione insufficiente a chi dovesse in seguito leggere il programma, e sono difficili da cambiare in maniera sistematica. Un modo per aggirare l'ostacolo è attribuire a costanti del genere dei nomi significativi. Una riga che inizi con `#define` stabilisce l'equivalenza tra una certa stringa di caratteri e un "simbolo di costante" (*symbolic constant*), che fungerà da lì in poi da nome per la stringa:

```
#define nome testo da sostituire
```

dove la stringa è indicata da *testo da sostituire*, e il simbolo di costante da *nome*. Ogni volta che *nome* apparirà nel seguito del programma (purché non tra virgolette o come parte di un altro nome), sarà sostituito dalla stringa *testo da sostituire*. L'espressione *nome* ha la stessa forma di un nome di variabile, e cioè una sequenza di lettere e numeri che inizi con una lettera, mentre *testo da sostituire* è una successione di caratteri arbitraria.

```
#include <stdio.h>

#define LOWER 0 /* valore minimo in gradi F nella tabella delle
               /* temperature */
#define UPPER 300 /* valore massimo in gradi F */
#define STEP 20 /* intervallo fra due temperature in gradi F
               /* adiacenti */

/* visualizza la tabella Fahrenheit-Celsius */
main()
{
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Le quantità `LOWER`, `UPPER` e `STEP` non sono variabili, ma simboli di costante, che quindi non figurano nelle dichiarazioni. I loro nomi sono scritti per convenzione in lettere maiuscole, cosicché è immediato distinguere dalle variabili, che sono invece convenzionalmente in lettere minuscole. Si noti che non c'è punto e virgola alla fine di una riga che inizi con `#define`.

## 1.5 Lettura e scrittura di caratteri

Passiamo ora a considerare una famiglia di programmi dedicati all'elaborazione di dati di tipo carattere. Il lettore avrà modo di scoprire che molti programmi non sono altro che versioni più accurate ed estese dei prototipi seguenti.

Il modello di ingresso e uscita dei dati, o input e output, adottato dalla libreria standard è molto semplice. Un testo, non importa da dove provenga o dove vada a finire, è trattato come una successione di caratteri. Un "flusso di testo" (*text stream*) è una sequenza di caratteri strutturata in righe; ogni riga si compone di zero o più caratteri seguiti dal carattere `\n` (*newline*). È compito della libreria far sì che tutti i dati in ingresso e uscita si conformino a questo modello: i programmatori C che utilizzano la libreria non devono preoccuparsi di come le righe siano rappresentate al di fuori del programma.

La libreria standard fornisce diverse funzioni per leggere o scrivere un carattere alla volta: `getchar` e `putchar` sono le più semplici. A ogni invocazione, `getchar` legge il *prossimo carattere in ingresso* da un flusso di testo e ne restituisce il valore. Ciò significa che dopo

```
c = getchar()
```

la variabile `c` conterrà il prossimo carattere dell'input. I caratteri in questione provengono, di norma, dalla tastiera; si parlerà della lettura da file nel Capitolo 7.

La funzione `putchar` visualizza un carattere per ogni chiamata:

```
putchar(c)
```

produce il contenuto della variabile intera `c` in forma di carattere, solitamente visualizzandolo sullo schermo. Si possono alternare le chiamate a `putchar` e a `printf`, ottenendo la visualizzazione del testo nel medesimo ordine delle chiamate.

### 1.5.1 Copia di file

Date `getchar` e `putchar`, è possibile scrivere una quantità sorprendente di codice senza sapere nulla l'altro su input e output. L'esempio di più immediata comprensione è un programma i cui dati in uscita siano la copia carattere per carattere dei suoi dati in ingresso:

*lettura di un carattere*

```
while (il carattere non è l'indicatore di fine del file)
    scrittura del carattere appena letto
    lettura di un carattere
```

La traduzione in C è:

```
#include <stdio.h>

/* rende i dati in uscita una replica dei dati in ingresso; prima versione */
main()
{
    int c;
    c = getchar();
```

```
        while (c != EOF) {
            putchar(c);
            c = getchar();
        }
    }
```

L'operatore relazionale `!=` significa "non eguale a".

Naturalmente, ciò che appare come carattere sullo schermo o sulla tastiera è rappresentato all'interno della macchina da una successione di bit, come per ogni altra entità manipolata da un calcolatore. Il tipo `char` è designato appositamente per la memorizzazione di questi caratteri, ma può essere usato qualunque tipo intero. Noi abbiamo scelto `int` per ovviare a un problema sottile e tuttavia importante, e cioè distinguere la fine dei dati in ingresso dai dati veri e propri.

La cosa è risolta grazie al fatto che, quando non vi sono più dati in ingresso, `getchar` restituisce un valore inconfondibile, che rimane distinto dagli altri caratteri: `EOF`, che sta per *end of file* ("fine del file"). Il tipo della variabile `c` deve ammettere valori sufficientemente grandi da contenere qualunque valore `getchar` restituisca. Non possiamo usare `char` poiché `c`, oltre ai caratteri ordinari per i quali `char` sarebbe sufficiente, deve poter contenere anche `EOF`: ecco perché usiamo `int`.

In concreto, `EOF` è un intero definito in `<stdio.h>`, ma il suo valore numerico non è rilevante purché non coincida con altri valori di `char`. Grazie all'uso del simbolo di costante, abbiamo la certezza che nulla nel programma dipende dal valore numerico specifico di `EOF`.

Un programmatore esperto scriverebbe il programma di replica dell'input più sinteticamente. In C ogni assegnamento, per esempio

```
c = getchar()
```

è un'espressione e dunque ha un valore, che dopo l'assegnamento corrisponde a quello del membro a sinistra. Ciò significa che un assegnamento può apparire come parte di un'espressione più complessa. Inserendo l'assegnamento di un carattere alla variabile `c` nella condizione per la terminazione di un ciclo `while`, il programma diviene:

```
#include <stdio.h>

/* rende i dati in uscita una replica dei dati in ingresso;
   seconda versione */
main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

L'iterazione `while` legge un carattere, lo assegna a `c`, quindi controlla se si tratti dell'indicazione di fine del file (`EOF`). Se non lo è, il corpo di `while` è eseguito, e il suo effetto è di visualizzare il carattere. L'iterazione continua finché viene raggiunta la fine dell'input; a questo punto hanno termine sia `while` che `main`.

Questa versione concentra le operazioni di lettura, lasciando un solo riferimento a `getchar`, e riduce il programma, che risulta più compatto e anche più leggibile, una volta acquisita una certa padronanza del linguaggio. Il lettore incontrerà spesso questo stile. (Il rischio però è di farsi prendere la mano e scivolare in una programmazione ermetica, tendenza che è opportuno tenere a bada.)

È necessario porre l'assegnamento tra parentesi all'interno della condizione di terminazione del ciclo. L'operatore `!=` ha *diritto di precedenza* su `=`, il che vuol dire che in mancanza delle parentesi la condizione relazionale `!=` sarebbe esaminata prima dell'assegnamento `=`, da cui segue che l'istruzione:

```
c = getchar() != EOF
```

è equivalente a

```
c = (getchar() != EOF)
```

Ne consegue l'effetto indesiderato di impostare il valore di `c` a 0 o a 1, a seconda dell'eventualità che chiamando `getchar` si incontri o meno la fine del file. La questione sarà approfondita nel Capitolo 2.

**Esercizio 1.6** Si verifichi che l'espressione `getchar() != EOF` abbia valore 0 o 1.

**Esercizio 1.7** Si scriva un programma che visualizzi il valore di `EOF`.

### 1.5.2 Conteggio dei caratteri

Il prossimo programma serve a contare i caratteri di un testo, ed è simile al programma di replica dei dati in ingresso appena analizzato.

```
#include <stdio.h>

/* conta i caratteri in ingresso; prima versione */
main()
{
    long nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

L'istruzione

```
++nc;
```

presenta un nuovo operatore, `++`, che va letto come “aumenta di uno il valore dell'operando”. Si potrebbe anche scrivere `nc = nc+1`, ma `++nc` è più conciso e si rivela spesso più efficiente. Esiste un operatore corrispondente `--` per diminuire di uno il valore dell'operando. Gli operatori `++` e `--` possono essere usati sia come prefissi (`++nc`) che come suffissi (`nc++`); queste due forme assumono valori differenti nelle espressioni, come si vedrà nel Capitolo 2,

ma entrambe hanno l'effetto di aumentare il valore di `nc` di un'unità. Per il momento ci limiteremo all'uso in forma di prefisso.

Il programma per il conteggio dei caratteri somma i valori in una variabile di tipo `long`, invece che `int`. Gli interi di tipo `long` misurano almeno 32 bit. Sebbene su alcune macchine `int` e `long` abbiano la stessa dimensione, su altre una variabile `int` è di 16 bit, con un valore massimo pari quindi a 32767, sicché anche un testo relativamente breve oltrepasserebbe il limite di un contatore di tale tipo. La specifica `%ld` chiarisce alla funzione `printf` che l'argomento corrispondente è un intero di tipo `long`.

A volte è possibile trattare numeri anche più grandi con il tipo `double` (`float` a doppia precisione). Useremo anche l'istruzione `for` invece di `while`, per esemplificare un altro modo di scrivere il ciclo.

```
#include <stdio.h>

/* conta i caratteri in ingresso; seconda versione */
main()
{
    double nc;

    for (nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

La funzione `printf` usa `%f` tanto per `float` che per `double`; `%.0f` evita la visualizzazione del punto decimale e della parte frazionaria, che è comunque zero.

Il corpo del ciclo `for` è vuoto, perché tutto il lavoro è fatto dalla valutazione della condizione e dall'esecuzione dell'incremento. Le regole grammaticali del C richiedono però che l'istruzione `for` abbia un corpo: il punto e virgola isolato, detto “istruzione nulla” (*null statement*), serve a tener fede a questa regola. Lo abbiamo disposto su una riga a sé per renderlo visibile.

Prima di chiudere con il programma per il conteggio dei caratteri, si noti che in mancanza di caratteri in ingresso la condizione per la terminazione del `while` o del `for` è falsa già alla prima chiamata di `getchar`, e il programma dà per risultato zero, cioè la risposta esatta. Una delle caratteristiche più utili di `while` e `for` è che essi valutano la condizione all'inizio del ciclo, prima di passare al corpo; se la condizione risulta subito falsa, il controllo passa alla prima istruzione dopo il corpo, con buona pace del suo contenuto. È bene che i programmi abbiano un comportamento ragionevole anche in assenza di input. Più in generale, le istruzioni `while` e `for` agevolano la stesura di codice in grado di comportarsi sensatamente anche laddove i dati in ingresso rappresentano casi estremi rispetto a ciò che ci si attende di norma.

### 1.5.3 Conteggio delle righe

Il prossimo programma permette di contare le righe di un testo. Come già accennato, la libreria standard fa sì che un flusso di testo in ingresso appaia come una sequenza di righe,

ognuna delle quali termina con il carattere `\n` (*newline*). Di conseguenza, per contare le righe basta contare le nuove righe:

```
#include <stdio.h>

/* conta il numero di righe del testo in ingresso */
main()
{
    int c, nl;

    nl = 0;
    while ((c=getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

Il contenuto del ciclo `while` consta ora di un'istruzione `if`, che a sua volta controlla l'incremento `++nl`. Il costrutto `if` verifica la condizione tra parentesi e, qualora sia vera, passa il controllo all'istruzione (o al gruppo di istruzioni tra parentesi graffe) che segue. Usiamo nuovamente i rientri per evidenziare la gerarchia del controllo.

Il segno di doppio uguale `==` è la notazione C per “è uguale a”, come il singolo `=` del Pascal o `.EQ.` del Fortran. Lo si usa per distinguere la condizione logica di egualanza dal singolo `=` che il C adopera per l'assegnamento. Un piccolo avvertimento per i neofiti del C: se si scrive `=` al posto di `==`, il risultato è di solito un'espressione lecita (cfr. Capitolo 2), ed è quindi facile che l'errore sfugga.

Un carattere scritto tra apici rappresenta un valore intero uguale al valore numerico del carattere nella rappresentazione interna della macchina. Si parla di *costante di tipo carattere*, malgrado sia solo un altro modo di scrivere un intero piccolo. Quindi '`A`', per esempio, è una costante di tipo carattere; nello standard ASCII ha valore 65, che è la rappresentazione interna del carattere A. Naturalmente la notazione '`A`' è preferibile a 65: il suo significato è ovvio ed è indipendente dalla rappresentazione interna.

Le sequenze di controllo ammesse nelle stringhe sono legittime anche nelle costanti di tipo carattere, e dunque '`\n`' rappresenta il valore del carattere *newline*, che è 10 in ASCII. Si badi bene che '`\n`' è un carattere singolo, e nelle espressioni rappresenta un singolo intero; "`\n`" è invece una costante stringa che ha la peculiarità di contenere un solo carattere. La tematica delle stringhe e dei caratteri sarà ulteriormente approfondita nel Capitolo 2.

**Esercizio 1.8** Si scriva un programma per contare gli spazi bianchi, i caratteri di tabulazione e i caratteri *newline*.

**Esercizio 1.9** Si scriva un programma i cui dati in uscita replichino i dati in ingresso, sostituendo però una stringa di uno o più spazi con uno spazio singolo.

**Esercizio 1.10** Si scriva un programma i cui dati in uscita replichino i dati in ingresso, sostituendo i caratteri di tabulazione con `\t`, ogni backspace con `\b` e ogni barra inversa con `\`, così da rendere perfettamente visibili le tabulazioni e le cancellazioni.

#### 1.5.4 Conteggio delle parole

Il quarto dei nostri programmi utili conta righe, parole e caratteri, dove per parola intendiamo una sequenza di caratteri che non contiene uno spazio, un carattere di tabulazione o un carattere *newline*. Questa è una versione ridotta all'osso del programma `wc` di UNIX.

```
#include <stdio.h>

#define IN 1 /* all'interno di una parola */
#define OUT 0 /* all'esterno di una parola */

/* conta il numero di righe, parole e caratteri del testo in ingresso */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c=getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Non appena incontra il carattere iniziale di una parola, il programma conta una nuova parola. La variabile `state` registra se il programma in un dato momento si trova o meno all'interno di una parola; all'inizio dell'esecuzione essa assume il valore `OUT`, che significa “all'esterno di una parola”. Ai valori letterali 1 e 0 preferiamo i simboli di costanti `IN` ed `OUT`, che rendono il programma più leggibile. Per programmi così brevi la differenza è trascurabile, ma in quelli più grandi il pregio di una maggior chiarezza compensa sicuramente lo sforzo di scrivere in tale maniera. Il lettore si accorgerà anche del fatto che è più semplice apportare cambiamenti di rilievo a programmi in cui i cosiddetti “numeri magici” compaiano solo come simboli di costanti.

La riga

```
nl = nw = nc = 0;
```

imposta a zero tutte e tre le variabili. Non si tratta di un caso particolare, ma è una conseguenza del fatto che un assegnamento è un'espressione con un valore e le regole di associazione degli assegnamenti procedono da destra a sinistra. È l'equivalente di:

```
nl = (nw = (nc = 0));
```

L'operatore `||` va letto OR, cioè denota la disgiunzione logica, per cui la riga

```
if (c == ' ' || c == '\n' || c == '\t')
```

significa: "se `c` è uno spazio bianco o `c` è un carattere newline o `c` è un carattere di tabulazione" (si rammenti che la sequenza di controllo `\t` è una rappresentazione esplicita del carattere di tabulazione). L'operatore corrispondente per la congiunzione è `&&` (AND), che ha precedenza rispetto a `||`. Le espressioni composte da congiunzioni (`&&`) o disgiunzioni (`||`) di altre espressioni si valutano da sinistra a destra, con la convenzione che la valutazione debba terminare non appena la verità o la falsità dell'intera espressione sia stata appurata. Se `c` è uno spazio bianco non è necessario sapere se è un carattere newline o un carattere di tabulazione, e perciò le relative condizioni non saranno esaminate. La cosa non ha per ora grande importanza, ma in contesti più complessi, come vedremo, ha effetti significativi.

In questo esempio c'è anche il costrutto `else`, che determina un'azione alternativa nel caso la condizione esaminata dall'istruzione `if` sia falsa. La forma generale è

```
if (espressione)
    istruzione1
else
    istruzione2
```

Sarà eseguita una e solo una delle due istruzioni. Se `espressione` è vera viene eseguita `istruzione1`, altrimenti viene eseguita `istruzione2`. Ciascuna `istruzione` può essere una singola istruzione o un gruppo di istruzioni delimitato da parentesi graffe. Nel programma per il conteggio delle parole il costrutto `if` che segue quello `else` controlla le due istruzioni tra parentesi graffe.

**Esercizio 1.11** In che modo si può mettere alla prova il programma per il conteggio delle parole? Quale tipo di dati in ingresso ha più probabilità di individuare eventuali bachi?

**Esercizio 1.12** Si scriva un programma che visualizzi il testo in ingresso una parola per riga.

## 1.6 Vettori

Scriviamo adesso un programma che conti il numero di ripetizioni di ogni data cifra e degli spazi bianchi (inclusi caratteri newline e tabulazioni), e conti anche gli altri caratteri. Questo problema, benché artificiale, ci consente di esaminare più aspetti del C in un solo programma.

Il programma prevede dodici diverse categorie di caratteri in ingresso, il che rende opportuno l'uso di un vettore per registrare il numero di occorrenze delle diverse cifre, piuttosto che dieci variabili individuali. Ecco una prima versione del programma:

```
#include <stdio.h>

/* conta il numero di cifre, spazi bianchi e altri caratteri del testo in
   ingresso*/
main()
```

```
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

    printf("cifre = ");
    for (i=0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", spazi bianchi = %d, altri = %d\n", nwhite, nother);
}
```

La dichiarazione

```
int ndigit[10];
```

specificava che `ndigit` è un vettore di dieci interi. In C, gli indici dei vettori partono sempre da zero, dunque gli elementi sono `ndigit[0]`, `ndigit[1]`, ..., `ndigit[9]`. Questo si riflette nelle due iterazioni (cicli `for`) che inizializzano e visualizzano il vettore.

Un indice può assumere la forma di qualunque espressione il cui valore sia un intero, incluse variabili come `i` e costanti intere.

Questo particolare programma si basa sulle proprietà della rappresentazione delle cifre tramite caratteri. Prendiamo per esempio la condizione

```
if (c >= '0' && c <= '9') ...
```

che stabilisce se il carattere in `c` è una cifra. In caso affermativo, il valore numerico di quella cifra è

```
c - '0'
```

La cosa funziona solo se '`0`', '`1`', ..., '`9`' hanno valori consecutivi crescenti, il che fortunatamente è vero per tutte le codifiche di caratteri; si veda per esempio una tabella ASCII.

Per definizione i valori del tipo `char` sono interi piccoli, e dunque il tipo `char` si comporta esattamente come il tipo `int` nelle espressioni aritmetiche. Questo è naturale e opportuno: per esempio, `c-'0'` è un'espressione il cui valore è un intero, compreso fra 0 e 9, che corrisponde al carattere tra '`0`' e '`9`' memorizzato nella variabile `c`, e perciò è un'indicazione legittima per il vettore `ndigit`.

La decisione che appura la natura di un carattere, se cioè sia una cifra, uno spazio o altro, è resa nel modo seguente.

```

if (c >= '0' && c <= '9')
    ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nwhite;
else
    ++nother;

```

La forma

```

if (condizione1)
    istruzione1
else if (condizione2)
    istruzione2
...
else ...
    istruzionen

```

ricorre spesso nei programmi come strumento per decidere fra più opzioni. Le *condizioni* sono valutate dall'alto verso il basso fino a che una di esse sia soddisfatta; l'*istruzione* corrispondente viene quindi eseguita, e l'esecuzione dell'intero costrutto è terminata. (Per *istruzione* si può sempre intendere un gruppo di istruzioni racchiuse tra parentesi graffe.) Se nessuna condizione è soddisfatta, l'*istruzione* che segue la clausola *else finale* è eseguita, nel caso sia presente; se invece non vi è alcuna clausola *else finale*, come nel programma di conteggio delle parole, non succede nulla: l'esecuzione dell'intero costrutto è terminata. Può esserci qualsiasi numero di clausole

```

else if (condizione)
    istruzione

```

tra la clausola *if* iniziale e quella *else finale*.

Per una questione di stile è consigliabile formattare questo costrutto nel modo indicato: se tutte le *if* fossero infatti rientrate a destra dopo la clausola *else* precedente, una lunga schiera di decisioni oltrepasserebbe il margine destro della pagina.

L'istruzione *switch*, che riprenderemo nel Capitolo 3, offre un altro modo per implementare la scelta fra diverse alternative, che si rivela particolarmente utile quando la condizione discriminante è la coincidenza del valore di un'espressione di tipo intero o carattere con un valore dello stesso tipo scelto da un insieme di costanti. Come termine di paragone, nel Paragrafo 3.4 presenteremo una versione del programma realizzata tramite *switch*.

**Esercizio 1.13** Si scriva un programma che visualizzi un istogramma della lunghezza delle parole contenute nel testo in ingresso. È facile disegnare un istogramma che si sviluppa in orizzontale, mentre richiede maggiore abilità tracciarlo in verticale.

**Esercizio 1.14** Si scriva un programma che visualizzi un istogramma delle frequenze dei diversi caratteri contenuti nel testo in ingresso.

## 1.7 Funzioni

In C una funzione corrisponde a una funzione nel Fortran, e a una procedura (o funzione) nel Pascal. Essa offre un metodo conveniente per racchiudere un brano di programma in grado di eseguire un compito specifico, in modo che sia riutilizzabile senza preoccuparsi della sua implementazione. Con funzioni progettate correttamente, si può ignorare *come* si ottiene un risultato: è sufficiente sapere *cosa* si ottiene. Il C rende l'uso delle funzioni semplice, comodo ed efficiente; si vedranno spesso funzioni brevi definite e chiamate una sola volta, allo scopo di rendere più chiaro un brano del codice.

Sinora abbiamo usato esclusivamente funzioni che già esistevano, quali *printf*, *getchar* e *putchar*; è giunto ora il momento di inventarne qualcuna. Visto che il C non ha operatori che eseguano il calcolo di una potenza come il *\*\** del Fortran, introduciamo il meccanismo di definizione delle funzioni scrivendo una funzione *power(m,n)* per elevare un intero *m* alla potenza intera positiva *n*. Per esemplificare, il valore di *power(2,5)* è 32. La funzione non è utilizzabile in applicazioni reali, poiché è in grado di calcolare solo potenze positive di interi piccoli, ma torna utile come esempio. (La libreria standard contiene una funzione *pow(x,y)* che calcola il valore di *x<sup>y</sup>*.)

Riportiamo sia la funzione *power* che il programma principale che la invoca, al fine di illustrare l'intera struttura del codice.

```

#include <stdio.h>

int power(int m, int n);

/* esegue un test della funzione power */
main()
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}

/* power: eleva la base alla n-esima potenza; n >= 0 */
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}

```

Una definizione di funzione assume questa forma:

```
tipo-restituito nome-funzione(dichiarazione dei parametri, se ve ne sono)
{
    dichiarazioni
    istruzioni
}
```

Le definizioni delle funzioni possono apparire in qualsiasi ordine e all'interno di uno o più file sorgente, benché nessuna di loro possa essere spezzata e suddivisa in file diversi. Se il programma sorgente è composto di più file, può darsi che siano necessari più interventi per compilarlo ed eseguirlo, ma è una questione che dipende dal sistema operativo, e non una prerogativa del linguaggio. Per adesso stabiliamo che entrambe le funzioni si trovino nello stesso file, così che il lettore possa applicare tutte le nozioni già apprese sull'esecuzione dei programmi scritti in C.

La funzione `power` è chiamata due volte nel corpo di `main`, alla riga

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Ogni invocazione passa due argomenti a `power`, che restituisce ogni volta un intero da formattare e visualizzare. All'interno di un'espressione, `power(2,i)` è un intero esattamente come lo sono `2` e `i`. (Non tutte le funzioni producono un valore intero; affronteremo l'argomento nel Capitolo 4.)

Già la prima riga di `power`,

```
int power(int base, int n)
```

dichiara i tipi e i nomi dei parametri, e il tipo del risultato restituito dalla funzione. I nomi che `power` usa per i suoi parametri sono circoscritti al suo corpo, e invisibili a ogni altra funzione; possono quindi essere riusati al di fuori di `power` senza generare conflitti. Questo vale anche per le variabili `i` e `p`: la `i` di `power` non è correlata alla `i` di `main`. Nomi (di variabili, parametri e così via) di questo genere si dicono *locali* (della funzione `power`, nel nostro esempio).

Diremo in genere *parametro* per indicare una variabile menzionata nell'elenco tra parentesi in una definizione di funzione, e *argomento* per il valore usato in una chiamata della funzione. Talvolta questi termini sono sostituiti rispettivamente dalle locuzioni *argomento formale* e *argomento attuale*.

Il valore calcolato da `power` è restituito a `main` dall'istruzione `return`. Qualunque espressione può seguire `return`:

```
return espressione;
```

Una funzione non deve necessariamente restituire un valore; l'istruzione

```
return;
```

chiede che il controllo torni al chiamante, senza che gli sia restituito alcun valore utile; ciò ha luogo anche quando, in assenza di un'istruzione `return`, la fine di una funzione è sancita semplicemente dalla parentesi graffa chiusa al termine del corpo. La funzione chiamante può ignorare il valore restituito da quella chiamata.

Il lettore avrà notato, nel nostro esempio, l'istruzione `return` alla fine della funzione `main`. Dato che `main` è una funzione come tutte le altre, può restituire un valore al chiamante, che è poi l'ambiente nel quale il programma è stato eseguito. Tipicamente la restituzione del valore zero indica la normale terminazione del programma, mentre valori diversi da zero indicano che le condizioni di terminazione sono errate o anomale. Per semplificare, finora l'istruzione `return` è stata omessa dalle funzioni `main`, ma da ora la includeremo: i programmi devono informare l'ambiente che li ospita dell'esito della loro esecuzione.

La dichiarazione

```
int power(int m, int n);
```

che precede `main` dice che la funzione `power` presuppone due argomenti interi e restituisce un intero. Questa dichiarazione prende il nome di *prototipo della funzione*, e deve concordare con la definizione di `power`: se ciò non avviene si incorre in un errore. I nomi dei parametri, però, non devono coincidere con quelli della definizione; anzi, la loro presenza nel prototipo è facoltativa. Avremmo potuto ugualmente scrivere:

```
int power(int, int);
```

Una buona scelta dei nomi, però, migliora la documentazione, e ne faremo spesso uso.

Una nota storica: il cambiamento più notevole tra la versione ANSI del C e quelle precedenti riguarda come dichiarare e definire le funzioni. Nella definizione originale del C, la funzione `power` avrebbe avuto questa struttura:

```
/* power: eleva la base alla potenza n-esima; n >= 0 */
/*          (versione vecchio stile)           */
power(base, n)
int base, n;
{
    int i, p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

I parametri compaiono tra parentesi, e i loro tipi sono dichiarati prima della parentesi graffa di apertura; i parametri non dichiarati sono considerati per default di tipo `int`. (Il contenuto della funzione è immutato.)

La dichiarazione di `power` all'inizio del programma avrebbe avuto questo aspetto:

```
int power();
```

Non era consentita una lista dei parametri, dunque il compilatore non poteva verificare agevolmente che `power` fosse chiamata nella maniera corretta. Addirittura, poiché in assenza di altre informazioni il compilatore avrebbe presupposto che `power` restituisse un intero, l'intera dichiarazione avrebbe potuto essere omessa.

La nuova sintassi dei prototipi semplifica il rilevamento di errori correlati al numero o ai tipi degli argomenti da parte del compilatore. Il vecchio modo di dichiarare e definire una

funzione rimarrà valido, almeno per un periodo di transizione, anche nel C ANSI, ma consigliamo vivamente di usare la nuova forma con i compilatori che la supportano.

**Esercizio 1.15** Si rielabori il programma di conversione delle temperature del Paragrafo 1.2 in modo da eseguire la conversione all'interno di una funzione.

## 1.8 Argomenti: chiamata per valore

Un aspetto delle funzioni del C può tornare poco familiare ai programmatore avvezzi ad altri linguaggi, il Fortran in particolare. Nel C tutti gli argomenti delle funzioni sono passati “per valore”. Questo vuol dire che i valori degli argomenti sono passati alla funzione chiamata mediante variabili provvisorie, anziché usare quelle originali. Ciò dà luogo ad alcune differenze rispetto alle “chiamate per riferimento” del Fortran o ai parametri dichiarati var del Pascal; in questi casi, infatti, la procedura chiamata ha accesso all'argomento originale, invece che a una copia locale. La differenza principale è che nel C la funzione chiamata non può modificare direttamente una variabile della funzione chiamante; può solo modificarne una copia temporanea e privata.

Tuttavia, la chiamata per valore non è una limitazione, ma una risorsa: permette di scrivere programmi più compatti, con meno variabili estranee o spurie, perché i parametri possono essere trattati dalla funzione chiamata alla stregua di variabili locali opportunamente inizializzate. Ecco una variante di power che sfrutta questa possibilità:

```
/* power: eleva la base alla potenza n-esima; n >= 0; versione 2 */
int power(int base, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

Il parametro n è usato come variabile provvisoria, ed è mano a mano decrementato da un ciclo for che va a ritroso finché non diventa zero; la variabile i non è più necessaria. Qualunque modifica si apporti a n all'interno di power non ha effetto sull'argomento con cui essa è stata originariamente invocata.

Quando richiesto, è possibile far sì che una funzione modifichi una variabile del chiamante. A tal fine, il chiamante dovrà fornire l'*indirizzo* della variabile da modificare (tecnicamente si parla del *puntatore* alla variabile), e la funzione chiamata dovrà dichiarare che il parametro è un puntatore: essa potrà così accedere alla variabile originale, per via indiretta, tramite il suo indirizzo. Tratteremo i puntatori nel Capitolo 5.

La situazione è diversa per i vettori. Quando il nome di un vettore è usato come argomento, il valore passato alla funzione è l'*indirizzo* del primo elemento del vettore: non vi è alcuna duplicazione dei suoi elementi. Tramite questo indirizzo, assieme a un appropriato indice, la funzione chiamata può accedere al vettore e modificarne un qualunque elemento. Questo è il tema del prossimo paragrafo.

## 1.9 Vettori di caratteri

Il tipo più comune di vettore in C è il vettore di caratteri. Per illustrarne l'uso, e mostrare come le funzioni operino sui vettori, scriveremo un programma che legga una serie di righe di testo e che poi visualizzi la più lunga. Lo schema è abbastanza semplice:

```
while (c'è un'altra riga)
    if (è più lunga della più lunga trovata finora)
        memorizza
            memorizza la sua lunghezza
    visualizza la riga più lunga
```

Da questo schema è chiaro che il programma si presta a essere diviso in segmenti. Un segmento legge una nuova riga, un altro la controlla, un altro la memorizza e l'ultimo dirige tutto il processo.

Stante questa naturale separazione dei compiti, sarebbe bene che anche il codice seguisse il medesimo ordine. Perciò scriveremo dapprima una funzione autonoma, `getline`, che legga la riga successiva del testo in ingresso. Tenteremo di rendere la funzione potenzialmente utile anche in altri contesti: come obiettivo minimo, `getline` deve segnalare l'eventuale fine del file (`EOF`); volendo progettarla in maniera più duttile, dovrebbe restituire anche la lunghezza della riga letta, o zero se non vi sono altri dati in ingresso. Zero è accettabile come valore convenzionale per la fine del file, perché non può denotare la lunghezza di alcuna riga: tutte le righe di testo, infatti, hanno almeno un carattere; persino una riga costituita da un solo newline ha lunghezza 1, dovendo contenere il carattere '`\n`'.

Quando si incontra una riga candidata a essere la più lunga, essa deve essere memorizzata; ciò suggerisce di scrivere una seconda funzione, che chiameremo `copy`, che serva a memorizzare la riga in un posto sicuro.

Infine, è necessario un programma principale che controlli `getline` e `copy`. Il risultato è mostrato di seguito.

```
#include <stdio.h>
#define MAXLINE 1000 /* massima lunghezza di una riga di input */

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* visualizza la riga più lunga del testo in ingresso */
main()
{
    int len;           /* lunghezza della riga corrente */
    int max;           /* lunghezza massima riscontrata finora */
    char line[MAXLINE]; /* riga corrente */
    char longest[MAXLINE]; /* contiene la riga più lunga trovata */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(longest, line);
        }
    if (max) {
        printf("la più lunga riga è %s", longest);
    }
}
```

```

        copy(longest, line);
    }
    if (max > 0) /* c'era almeno una riga in ingresso */
        printf("%s", longest);
    return 0;
}

/* getline: legge una riga in ingresso, la assegna a s, ne restituisce
   la lunghezza */
int getline(char s[], int lim)
{
    int c, i;

    for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}

/* copy: copia la stringa 'from' sulla stringa 'to'; assume che 'to' sia
   sufficientemente lunga */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}

```

Le funzioni `getline` e `copy` sono dichiarate all'inizio del programma, che assumiamo essere contenuto in un solo file.

Le funzioni `main` e `getline` interagiscono grazie a una coppia di argomenti e a un valore restituito. In `getline`, gli argomenti sono dichiarati da

```
int getline(char s[], int lim)
```

dove si specifica che il primo argomento, `s`, è un vettore, e il secondo, `lim`, è un intero. La dichiarazione di un vettore include la sua dimensione, cosicché il sistema possa allocare la memoria necessaria e sufficiente a contenerlo. La lunghezza del vettore `s` nell'intestazione di `getline` non è richiesta, dato che l'allocazione avviene in `main`. La funzione `getline` usa `return` per restituire un valore a `main`, proprio come faceva la funzione `power`. La definizione di `getline` dichiara anche che essa restituisce un intero; visto che, in assenza di altre informazioni, si presuppone che il tipo restituito da una funzione sia `int`, tale dichiarazione può essere omessa.

Alcune funzioni restituiscono un valore utile; di altre, come `copy`, interessa solo che eseguano un certo compito senza restituire alcun valore. Ciò è palesemente evidente dalla definizione di `copy`, il cui tipo è dichiarato `void` (cioè vuoto, nullo).

La funzione `getline` pone il carattere '`\0`' (il "carattere nullo", o *null character*), il cui valore è zero, alla fine del vettore che sta creando, per segnalare la fine della stringa di caratteri. Questa è una convenzione generale del C: quando una costante stringa come

```
"ciao\n"
```

appare in un programma in C, è conservata sotto forma di un vettore di caratteri che contiene i caratteri della stringa seguiti da '`\0`' per indicarne la fine.

c	i	a	o	<code>\n</code>	<code>\0</code>
---	---	---	---	-----------------	-----------------

La specifica `%s` all'interno di `printf` prescrive che l'argomento corrispondente sia una stringa rappresentata in questa forma. Anche `copy` si aspetta che la stringa in ingresso termini con '`\0`', e si cura di trascrivere questo carattere nel doppione che genera. (Il tutto implica che '`\0`' non faccia parte del testo normale.)

Per inciso, vale la pena notare che persino un programma così piccolo presenta degli aspetti progettuali spinosi. A titolo di esempio: cosa dovrebbe fare `main` se incontrasse una riga più lunga del limite prestabilito? In effetti `getline` è una funzione sicura, nel senso che interrompe la lettura quando il vettore è pieno, anche se non ha incontrato alcun carattere newline. Analizzando la lunghezza e l'ultimo carattere restituito, `main` può stabilire se l'ultima riga sia troppo lunga, e trovare una soluzione. Per non dilungarci, abbiamo ignorato la questione.

Non esiste, per chi usa `getline`, un modo di sapere in anticipo quanto potrà risultare lunga una riga in ingresso, per cui è la medesima `getline` a esercitare i controlli necessari. Viceversa, chi usa `copy` conosce già (o può scoprire) la dimensione delle stringhe: abbiamo quindi scelto di non aggiungere la gestione degli errori a `copy`.

**Esercizio 1.16** Si riveda la funzione `main` del programma presentato in questa sezione, affinché visualizzi correttamente la lunghezza di righe arbitrariamente lunghe, e ne mostri la maggior parte di testo possibile.

**Esercizio 1.17** Si scriva un programma che visualizzi tutte le righe in ingresso che superino gli 80 caratteri.

**Esercizio 1.18** Si scriva un programma per eliminare spazi e tabulazioni all'inizio e alla fine di ogni riga in ingresso e per eliminare le righe completamente vuote.

**Esercizio 1.19** Si scriva una funzione `reverse(s)` che capovolga la stringa `s`. La si usi poi per scrivere un programma che capovolga, una per volta, le righe del testo in ingresso.

## 1.10 Variabili esterne e visibilità

Le variabili di `main`, quali `line`, `longest` e così via, sono dette private o locali di `main`. Poiché sono dichiarate all'interno di `main`, nessun'altra funzione può accedervi direttamente. Lo stesso principio vale per le variabili delle altre funzioni; così, la variabile `i` in `getline` non è

correlata alla *i* di copia. Le variabili locali di una funzione si materializzano solo nel momento in cui essa viene invocata, per poi svanire quando essa termina. Per questo motivo sono dette comunemente variabili *automatiche*; adotteremo nel prosieguo questa terminologia, propria anche di altri linguaggi. (Il Capitolo 4 espone la classe di memorizzazione statica, nella quale le variabili locali possono mantenere i propri valori tra una chiamata e l'altra.)

Siccome le variabili automatiche appaiono e scompaiono in dipendenza dalle chiamate della funzione, esse non possono mantenere i loro valori tra una chiamata e l'altra, e devono essere esplicitamente impostate a ogni chiamata; se ciò non avviene, conterranno materiale di scarto.

In alternativa, è possibile definire variabili *esterne* a tutte le funzioni, ovvero accessibili da parte di qualsiasi funzione tramite il loro nome. (Il meccanismo è piuttosto simile al COMMON del Fortran o al funzionamento in Pascal delle variabili dichiarate nel blocco più esterno.) Vista la loro accessibilità globale, le variabili esterne possono sostituire le liste di argomenti per lo scambio di dati tra funzioni. Inoltre, poiché le variabili esterne hanno durata permanente, invece di apparire e scomparire in base alle chiamate delle funzioni, mantengono i loro valori anche dopo che le funzioni che le hanno modificate hanno smesso di operare.

Una variabile esterna deve essere *definita*, una e una sola volta, al di fuori di qualunque funzione; la definizione porta il sistema a riservare la quantità appropriata di memoria per i contenuti della variabile. La variabile deve anche essere *dichiarata* in ogni funzione che intenda accedervi; ciò esplicita il tipo della variabile. La dichiarazione può essere un'istruzione *extern*, o risultare implicitamente dal contesto. Per dare concretezza alla discussione, riscriviamo il programma per la ricerca della riga di testo più lunga impostando *line*, *longest* e *max* come variabili esterne; sarà necessario modificare le chiamate, le dichiarazioni e i contenuti delle tre funzioni.

```
#include <stdio.h>

#define MAXLINE 1000      /* massima lunghezza di una riga di input */
int max;                  /* lunghezza massima riscontrata finora */
char line[MAXLINE];      /* riga corrente */
char longest [MAXLINE];   /* contiene la riga più lunga trovata */

int getline(void);
void copy(void);

/* visualizza la riga più lunga del testo in ingresso; versione
   specializzata */
main()
{
    int len;
    extern int max;
    extern char longest[];
```

```
max = 0;
while ((len = getline()) > 0)
    if (len > max) {
        max = len;
        copy();
    }
if (max > 0)    /* c'era almeno una riga in ingresso */
    printf("%s", longest);
return 0;
}

/* getline: versione specializzata */
int getline(void)
{
    int c, i;
    extern char line[];

    for (i = 0; i < MAXLINE-1 && (c=getchar()) != EOF && c != '\n'; ++i)
        line[i] = c;
    if (c == '\n') {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return i;
}

/* copy: versione specializzata */
void copy(void)
{
    int i;
    extern char line[], longest[];

    i = 0;
    while ((longest[i] = line[i]) != '\0')
        ++i;
}
```

Le variabili esterne di *main*, *getline* e *copy* sono definite nelle prime righe del programma, che ne dichiarano il tipo e provocano l'opportuna allocazione di memoria. Sintatticamente le definizioni esterne sono affini alle definizioni delle variabili locali. Prima che una funzione possa usare una variabile esterna, il nome della variabile deve essere reso noto alla funzione. Per sciogliere questo nodo si può introdurre nella funzione la dichiarazione *extern*; la parte rimanente della dichiarazione è invariata.

In determinate circostanze, la dichiarazione `extern` può essere omessa. Se la definizione di una variabile esterna cade in un file sorgente prima di essere usata in una certa funzione, non è necessario inserirla nella funzione; le dichiarazioni `extern` in `main`, `getline` e `copy` sono quindi ridondanti. È piuttosto comune collocare le definizioni di tutte le variabili esterne all'inizio del file sorgente, e poi omettere tutte le dichiarazioni `extern`.

Ipotizziamo che il programma sia suddiviso in diversi file sorgente: se una variabile è definita nel `file1` ma viene usata nel `file2` e nel `file3`, è necessario che il `file2` e il `file3` contengano una dichiarazione `extern`, per poter correlare le occorrenze della variabile. È prassi raccomandare le dichiarazioni `extern` relative a variabili e funzioni in un file distinto, detto “intestazione” (*header*). L'intestazione viene richiamata all'inizio di ogni file sorgente mediante l'istruzione `#include`, e al nome di tali file di intestazione è attribuito il suffisso convenzionale `.h`. Le funzioni della libreria standard, per esempio, sono dichiarate in intestazioni come `<stdio.h>`. Si discuterà di questo argomento per esteso nel Capitolo 4, mentre la libreria è trattata nel Capitolo 7 e nell'Appendice B.

Poiché le versioni specializzate di `getline` e `copy` non contengono argomenti, la logica suggerirebbe che i loro prototipi all'inizio del file siano `getline()` e `copy()`. Per garantire la compatibilità con programmi in C del passato, però, lo standard interpreta una lista vuota come una dichiarazione alla vecchia maniera e deroga al controllo degli argomenti della lista; non si incorre in tale deroga solo indicando esplicitamente una lista vuota con `void`, come vedremo nel Capitolo 4.

Si osservi la cautela esercitata nell'uso delle parole *definizione* e *dichiarazione* in riferimento a variabili esterne. “Definizione” è inerente al luogo di creazione della variabile, con relativa allocazione di memoria; “dichiarazione” si riferisce ai luoghi dove è asserita la natura della variabile, ma senza che sia riservata memoria.

Va detto che si tende ad abusare del costrutto `extern`, perché sembra facilitare la comunicazione tra funzioni: le liste di argomenti sono più brevi e le variabili restano sempre a disposizione. Purtroppo, però, le variabili esterne sono sempre lì, anche quando non le si vorrebbe. Fare troppo affidamento sulle variabili esterne è rischioso, perché si possono generare programmi in cui il flusso delle informazioni è tutt'altro che chiaro: le variabili possono subire cambiamenti imprevisti e persino involontari, e il programma è difficile da modificare. La seconda versione del nostro programma è di livello inferiore alla prima, in parte per questi motivi e in parte perché annienta il carattere generale di due utili funzioni, cancellando in modo permanente al loro interno i nomi delle variabili che usano.

A questo punto abbiamo esaurito ciò che si può considerare il nucleo tradizionale del C. Con questa dotazione di elementi base è possibile scrivere utili programmi di buone dimensioni, e sarebbe un'idea sensata fermarsi per raggiungere questo obiettivo. Gli esercizi che seguono si riferiscono a programmi in certa misura più complessi di quelli finora affrontati.

**Esercizio 1.20** Si scriva un programma `datab` che sostituisca i caratteri di tabulazione nel testo in ingresso con il numero di spazi bianchi equivalente. Si assuma un intervallo di tabulazione predeterminato, per esempio ogni  $n$  colonne. Che cosa dovrebbe essere  $n$ , una variabile o un parametro simbolico?

**Esercizio 1.21** Si scriva un programma `intab` che rimpiazzi le stringhe di spazi bianchi con il numero minimo di caratteri di tabulazione e di singoli spazi necessario per ottenere la medesima spaziatura. Si usi lo stesso intervallo di tabulazione del programma `datab`. Quando sia una tabulazione che un singolo spazio sono sufficienti per raggiungere la spaziatura desiderata, quale dei due è da preferire?

**Esercizio 1.22** Si scriva un programma che spezzi le righe in ingresso troppo lunghe in due o più righe brevi; la riga deve essere interrotta dopo l'ultimo carattere diverso da uno spazio che occorre prima dell' $n$ -esima colonna di testo in ingresso. Il programma si dovrebbe comportare in maniera intelligente in presenza di righe molto lunghe, o quando non vi siano spazi o caratteri di tabulazione prima della colonna specificata.

**Esercizio 1.23** Si scriva un programma che rimuova tutti i commenti da un programma in C. Non si trascuri di gestire correttamente le stringhe tra virgolette e le costanti carattere tra apici. Il C non permette di annidare i commenti.

**Esercizio 1.24** Si scriva un programma per rilevare grossolani errori di sintassi dei programmi in C, per esempio l'incoerenza fra parentesi (tonde, quadre e graffe) aperte e chiuse. Non si tralascino gli apici e le virgolette, le sequenze di controllo e i commenti. (Nella sua versione più generale questo è un programma alquanto difficile.)

## Tipi, operatori ed espressioni

VARIABILI E COSTANTI costituiscono le principali categorie di dati utilizzate da un programma. Le dichiarazioni elencano le variabili da usare, ne asseriscono il tipo e talvolta i valori iniziali, mentre gli operatori specificano che cosa si deve fare con esse. Le espressioni combinano variabili e costanti per produrre nuovi valori. Il tipo di un oggetto determina la gamma di valori che può assumere e le operazioni ammesse su di esso. Il capitolo è incentrato su questi elementi di base.

Lo standard ANSI ha aggiunto molte modifiche ai tipi fondamentali e alle espressioni. Tutte le varianti degli interi contemplano ora versioni con e senza segno (`signed` e `unsigned`), e sono disponibili notazioni specifiche per le costanti prive di segno e quelle esadesimali. Le operazioni con virgola mobile possono essere eseguite con precisione singola; per la precisione multipla si può ricorrere al tipo `long double`. Le costanti stringa possono essere concatenate al momento della compilazione. Le enumerazioni sono entrate a far parte del linguaggio, istituzionalizzando una caratteristica da lungo tempo presente in via informale. La dichiarazione `const` rende il proprio oggetto non modificabile. Le regole per le conversioni automatiche fra tipi aritmetici sono state arricchite per far fronte all'ampliata gamma di tipi.

## 2.1 Nomi delle variabili

Anche se non l'abbiamo detto nel Capitolo 1, esistono alcune restrizioni rispetto ai nomi di variabili e simboli di costanti. I nomi sono composti da lettere e cifre e devono cominciare con una lettera. Il carattere di sottolineatura “\_” (*underscore*) vale come lettera, e talvolta è utile per migliorare la leggibilità dei nomi lunghi; tuttavia, non deve trovarsi all'inizio del nome di una variabile, perché le funzioni della libreria spesso usano questo artificio. Si badi a distinguere minuscole e maiuscole; x e X sono due nomi diversi. In C, per convenzione, si adoperano le minuscole per i nomi delle variabili e le maiuscole per i simboli di costanti.

Sono significativi come minimo i primi 31 caratteri di un nome locale. Il numero potrebbe diminuire per i nomi delle funzioni e le variabili esterne, poiché i nomi globali possono essere usati da assemblatori e caricatori sui quali il linguaggio non ha alcun controllo. Quanto ai nomi esterni, lo standard ne garantisce l'univocità limitatamente a 6 caratteri, senza distinzione fra maiuscole e minuscole. Parole chiave come if, else, int, float e così via sono appannaggio esclusivo del linguaggio: non possono figurare come nomi di variabili e devono essere scritte in lettere minuscole.

È sensato associare alle variabili nomi attinenti al loro scopo e facili da scrivere senza errori di battitura. Consigliamo di dare nomi brevi alle variabili locali interne, specialmente agli indici dei cicli, e nomi più lunghi a quelle esterne.

## 2.2 Tipi e dimensioni dei dati

Il C prevede un numero ristretto di tipi di dati fondamentali.

<code>char</code>	un singolo byte, in grado di contenere un carattere dell'ambiente in cui risiede il compilatore.
<code>int</code>	un intero, solitamente pari alla dimensione naturale degli interi sulla macchina in cui risiede il compilatore.
<code>float</code>	numero con virgola mobile, precisione singola.
<code>double</code>	numero con virgola mobile, precisione doppia.

In aggiunta, è possibile qualificare questi tipi in vari modi. Gli attributi `short` e `long` si applicano agli interi:

```
short int sh;
long int counter;
```

Il termine `int` può essere omesso da queste dichiarazioni, come normalmente avviene.

L'idea è che `short` e `long` denotino interi di diverse lunghezze, laddove praticabile; `int` è invece di solito la grandezza naturale di riferimento per una data macchina. Per `short` si hanno spesso 16 bit, per `long` 32 e per `int` 16 o 32. Ogni compilatore può liberamente scegliere le dimensioni che ritiene opportune per la macchina; le uniche limitazioni impongono che gli interi `short` e `int` siano di almeno 16 bit, quelli `long` di almeno 32, e che la dimensione di un intero `short` non superi quella di un `int`, la quale a sua volta non può superare quella di un `long`.

L'attributo `signed` (con segno) o `unsigned` (privo di segno) può essere applicato a `char` o a qualunque intero. I numeri `unsigned` sono sempre positivi o al più nulli, e obbediscono alle leggi dell'aritmetica modulo  $2^n$ , dove  $n$  è il numero di bit del tipo. Quindi, se per esempio `char` è di 8 bit, il valore delle variabili `unsigned char` oscilla tra 0 e 255 e quello delle variabili `signed char` tra -128 e 127 (se la macchina adotta il complemento a due). Dipende dalla macchina se i `char` semplici siano con o senza segno, ma i caratteri stampabili hanno sempre valore positivo.

Il tipo `long double` denota numeri con virgola mobile a precisione multipla. Come per gli interi, la dimensione degli oggetti a virgola mobile è definita dall'implementazione; i tipi `float`, `double` e `long double` potrebbero denotare una, due o tre dimensioni distinte.

Le intestazioni standard `<limits.h>` e `<float.h>` contengono simboli di costanti che descrivono tutte queste grandezze, e altre proprietà legate alla macchina e al compilatore analizzate nell'Appendice B.

**Esercizio 2.1** Si scriva un programma che determini la gamma di valori possibili dei tipi `char`, `short`, `int` e `long`, con o senza segno, per calcolo diretto o visualizzando valori appropriati dalle intestazioni standard. Un esercizio più difficile, se eseguito per calcolo diretto: si stabilisca la gamma di valori dei diversi tipi con virgola mobile.

## 2.3 Costanti

Una costante intera quale 1234 è di tipo `int`. Una costante `long` si scrive con una `l` (elle) o `L` finale, come in `123456789L`; un intero troppo grande per essere `int` sarà automaticamente considerato `long`. Le costanti senza segno sono scritte con una `u` o `U` finale, mentre il suffisso `ul` o `UL` indica il tipo `unsigned long`.

Le costanti con virgola mobile contengono il punto decimale (123.4) oppure un esponente (1e-2), o entrambi; il loro tipo è `double`, a meno che rechino un suffisso. I suffissi `f` o `F` indicano una costante di tipo `float`, `l` o `L` di tipo `long double`.

Un intero può anche essere scritto in base 8 (notazione ottale) o in base 16 (notazione esadecimale), invece che nell'ordinaria base 10. Uno zero all'inizio di una costante intera significa notazione ottale; il simbolo `0x` o `0X`, anch'esso in testa al numero, denota base 16. Per esempio, 31 (numerazione decimale) può essere scritto come `037` in ottale e come `0x1f`, o anche `0X1F`, in esadecimale. Alle costanti ottali ed esadecimali è possibile far seguire `l` per renderle di tipo `long` e `u` per renderle `unsigned`: `0XFUL` è una costante di tipo `unsigned long`; in notazione decimale è il numero 15.

Una costante (di tipo) `carattere` è un intero, scritto come un carattere tra apici: 'x' ne è un esempio. Il valore di una costante carattere è il valore numerico corrispondente nel set dei caratteri della macchina. In riferimento al set di caratteri ASCII, la costante carattere '`0`' ha valore 48, diverso dal valore numerico `0`. Scrivendo '`0`' anziché un valore numerico dipendente dal set di caratteri, come 48, il programma può prescindere dall'effettivo valore e risulta più facile da leggere. Le costanti carattere prendono parte alle operazioni numeriche come tutti gli altri interi, anche se per lo più le si usa in confronto con altri caratteri.

Nelle costanti stringa e carattere, alcuni caratteri possono essere rappresentati da sequenze di controllo come `\n` (carattere *newline*); le sequenze sono composte da due caratteri che ne rappresentano uno solo. Una qualunque configurazione di bit delle dimensioni di un byte si può denotare così:

`'\ooo'`

dove `ooo` rappresenta da una a tre cifre in ottale (0...7), oppure così:

`'\xhh'`

dove `hh` è una o più cifre esadecimale (0...9, a...f, A...F). Dunque potremmo scrivere

```
#define VTAB '\013' /* tabulazione verticale ASCII */
#define BELL '\007' /* segnale d'allerta ASCII */
```

o, in notazione esadecimale,

```
#define VTAB '\xb' /* tabulazione verticale ASCII */
#define BELL '\x7' /* segnale d'allerta ASCII */
```

L'elenco completo di sequenze di controllo è il seguente:

<code>\a</code>	segnale d'allerta (campanello)	<code>\\\</code>	barra inversa
<code>\b</code>	backspace	<code>\?</code>	punto interrogativo
<code>\f</code>	salto pagina	<code>\'</code>	apice
<code>\n</code>	carattere newline	<code>\"</code>	virgolette
<code>\r</code>	ritorno del carrello	<code>\ooo</code>	numero in base otto
<code>\t</code>	tabulazione orizzontale	<code>\xhh</code>	numero in base sedici
<code>\v</code>	tabulazione verticale		

La costante carattere `'\0'` rappresenta il carattere di valore zero, o carattere nullo; nelle espressioni di tipo carattere, è spesso preferito a `0` per rimarcarne la natura, ma il suo valore è comunque zero.

Un'espressione costante è un'espressione che contiene solo costanti. Tali espressioni sono valutabili in fase di compilazione anziché di esecuzione, e quindi possono tranquillamente essere inserite senza tema d'inefficienze ovunque sia ammessa una costante, come in

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

o

```
#define LEAP 1 /* negli anni bisestili */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Una costante (di tipo) stringa o letterale (di tipo) stringa è una sequenza di zero o più caratteri racchiusi tra virgolette, come in

```
"Sono una stringa"
o
"" /* la stringa vuota */
```

Le virgolette non fanno parte della stringa, ma servono solo a delimitarla. Le stesse sequenze di controllo appena viste per le costanti carattere si applicano alle stringhe; `\"` è il

segno delle doppie virgolette. Le costanti stringa possono essere concatenate in fase di compilazione:

`"ciao, " "mondo"`

equivale a

`"ciao, mondo"`

Questo torna utile per spezzare le stringhe lunghe e distribuirle su più righe di codice.

Tecnicamente una costante stringa è un vettore di caratteri. La rappresentazione interna delle stringhe termina con un carattere nullo `'\0'`, cosicché la quantità di memoria necessaria per contenere una stringa è data dal numero di caratteri tra virgolette aumentato di uno. Questo modo di rappresentare le stringhe significa che non esiste un limite predefinito alla loro lunghezza, ma anche che i programmi devono scandire per intero se vogliono determinarla. La funzione `strlen(s)` della libreria standard restituisce la lunghezza della stringa `s`, a esclusione del `'\0'` finale. Questa è la nostra versione:

```
/* strlen: restituisce la lunghezza di s */
int strlen(char s[])
{
    int i;
    i = 0;
    while (s[i] != '\0')
        ++i;
    return i;
}
```

La funzione `strlen` e altre relative alle stringhe sono dichiarate nell'intestazione standard `<string.h>`.

Si faccia attenzione a distinguere tra una costante di tipo carattere e una stringa con un solo carattere: `'x'` non è uguale a `"x"`. Il primo è un intero, e denota il valore numerico della lettera `x` nel set di caratteri della macchina. Il secondo è un vettore di caratteri contenente un carattere (la lettera `x`) e uno `'\0'`.

Esiste un altro tipo di costante, quella di *enumerazione*. Si tratta di una lista di valori interi costanti, come in

```
enum boolean { NO, YES };
```

Il primo nome in un'enumerazione ha valore 0, il seguente 1 e così via, se non vengono dati esplicitamente altri valori. Qualora rimangano valori non specificati, essi continuano la progressione a partire dall'ultimo valore specificato, come nel secondo degli esempi seguenti:

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };
```

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
JUL, AUG, SEP, OCT, NOV, DEC }
/* FEB è 2, MAR è 3, ecc. */
```

I nomi in enumerazioni differenti *devono* essere distinti, mentre nella stessa enumerazione i valori possono anche ripetersi.

Le enumerazioni forniscono una comoda tecnica alternativa a `#define` per associare nomi a costanti, che ha anche il vantaggio di poter generare valori automaticamente. Sebbene le variabili `enum` possano essere dichiarate, i compilatori non sono obbligati a verificare che il loro contenuto sia un valore valido. Cionondimeno, le variabili di enumerazione offrono la possibilità della verifica, e quindi sono preferibili a `#define`. A ciò si aggiunga che un debugger può essere in grado di visualizzare i valori delle variabili di tipo enumerazione nella loro forma simbolica.

## 2.4 Dichiarazioni

Tutte le variabili devono essere dichiarate prima di essere usate, benché alcune dichiarazioni possano essere implicite nel contesto. Una dichiarazione specifica un tipo, e contiene un elenco di una o più variabili di quel tipo, per esempio

```
int lower, upper, step;
char c, line[1000];
```

È possibile distribuire a piacimento le variabili tra le dichiarazioni; avremmo potuto scrivere con uguale efficacia:

```
int lower;
int upper;
int step;
char c;
char line[1000];
```

Quest'ultima forma occupa più spazio, ma si presta bene all'aggiunta di un commento a ciascuna dichiarazione, o a modifiche successive.

È anche possibile inizializzare una variabile nella sua dichiarazione. Se il nome è seguito da un segno di uguale e da un'espressione, questa funge da inizializzatore, come in

```
char esc = '\\';
int i = 0;
int limit = MAXLINE+1;
float eps = 1.0e-5;
```

Se la variabile in questione non è automatica, l'inizializzazione avviene una volta sola, concettualmente prima dell'inizio dell'esecuzione del programma, e l'inizializzatore deve essere un'espressione costante. Una variabile automatica esplicitamente inizializzata è inizializzata ogni volta che si accede alla funzione o al blocco che la contiene; qualunque espressione può fungere da inizializzatore. Le variabili statiche e quelle esterne sono inizializzate a zero per default. Le variabili automatiche per le quali non esiste un inizializzatore esplicito hanno valori indefiniti: non è dato cioè sapere quale valore contengano.

L'attributo `const` può essere applicato alla dichiarazione di qualsiasi variabile, con l'effetto di prescrivere che il suo valore non cambierà. Nel caso di un vettore, `const` afferma che gli elementi non subiranno modifiche.

```
const double e = 2.71828182845905;
const char msg[] = "attenzione: ";
```

L'attributo `const` può anche essere applicato ad argomenti vettoriali, nel qual caso indica che la funzione non cambia tale vettore:

```
int strlen(const char[]);
```

L'effetto di un tentativo di modifica di un oggetto `const` è definito dall'implementazione.

## 2.5 Operatori aritmetici

Gli operatori aritmetici binari sono `+`, `-`, `*`, `/` e l'operatore modulo `%`. La divisione tra interi tronca la parte frazionaria. L'espressione

```
x % y
```

dà per risultato il resto della divisione, e quindi dà zero se `y` divide `x` esattamente. Per esempio, un anno è bisestile se lo si può dividere per 4 ma non per 100, salvo gli anni divisibili per 400, che sono bisestili. Di conseguenza:

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    printf("%d e' un anno bisestile\n", year);
else
    printf("%d non e' un anno bisestile\n", year);
```

L'operatore `%` non può essere applicato a valori `float` o `double`. Per gli interi negativi, la direzione di troncamento di `/` e il segno del risultato di `%` dipendono dalla macchina, come anche le conseguenze del traboccamento (*overflow* e *underflow*).

Gli operatori binari `+ e -` hanno lo stesso grado di precedenza, inferiore a quello di `*`, `/` e `%`, che a sua volta è inferiore a quello degli operatori unari `+ e -`.

Gli operatori aritmetici si raggruppano da sinistra a destra.

La Tabella 2.1 alla fine del capitolo riassume le regole di associatività e il grado di precedenza per tutti gli operatori.

## 2.6 Operatori relazionali e logici

Gli operatori relazionali sono

```
>     >=    <     <=
==    !=
```

e hanno tutti pari precedenza. Hanno grado di precedenza appena più basso gli operatori di eguaglianza:

```
==    !=
```

Gli operatori aritmetici hanno diritto di precedenza su quelli relazionali, e dunque l'espressione `i < lim-1` si legge `i < (lim-1)`, com'è d'altronde naturale.

Più interessanti sono gli operatori logici `&&` e `||`. Le espressioni contenenti `&&` o `||` sono valutate da sinistra a destra; la valutazione si interrompe non appena il risultato si rivela vero o falso. Molti programmi in C sfruttano questa proprietà: a titolo di esempio, ecco un ciclo dalla funzione `getline` che abbiamo scritto nel Capitolo 1:

```
for (i=0; i<lim-1 && (c=getchar()) != EOF && c != '\n'; ++i)
    s[i] = c;
```

Prima di leggere un nuovo carattere è indispensabile verificare che ci sia spazio per memorizzarlo nel vettore `s`, per cui la condizione `i < lim-1` deve essere esaminata *prima*. Inoltre, se la condizione è falsa occorre interrompere la lettura dei caratteri.

In maniera analoga, sarebbe ben poco opportuno confrontare `c` con `EOF` prima di chiamare `getchar`, e quindi chiamata e assegnamento devono avvenire prima che il carattere in `c` sia esaminato.

L'operatore `&&` ha diritto di precedenza su `||`, ed entrambi danno precedenza agli operatori relazionali e di egualanza; dunque espressioni come

```
i<lim-1 && (c = getchar()) != EOF && c != '\n'
```

non richiedono altre parentesi, ma visto che `!=` ha diritto di precedenza sull'assegnamento, le parentesi sono necessarie in

```
(c=getchar()) != EOF
```

per ottenere il risultato voluto: l'assegnamento a `c`, e solo in seguito il confronto con `EOF`.

Per definizione, il valore numerico di un'espressione relazionale o logica è 1 se la relazione è vera, e zero se la relazione è falsa.

L'operatore unario di negazione `!` trasforma un numero diverso da zero in 0, e lo zero in 1. Un utilizzo frequente di `!` si ha in costrutti come

```
if (!valid)
```

piuttosto che

```
if (valid == 0)
```

È difficile dire, in generale, quale sia la forma migliore. Costrutti semplici come `!valid` sono di agevole lettura (*if not valid*, ovvero "se non è valid"), ma combinazioni più complesse sono di ardua comprensione.

**Esercizio 2.2** Si scriva un ciclo equivalente a quello `for` presentato in questo paragrafo, senza usare `&&` o `||`.

## 2.7 Conversioni di tipo

Quando un operatore ha operandi di tipo diverso, essi sono convertiti nello stesso tipo secondo alcune regole. In genere, le uniche conversioni automatiche sono quelle che consentono a un tipo di ampliare il proprio campo di variabilità senza perdita di informazione, come succede trasformando un intero in un numero con virgola mobile in un'espressione

quale `f + i`. Espressioni prive di senso, come numeri `float` in qualità di indice, non sono tollerate. Espressioni che potrebbero causare perdita d'informazione, come assegnare un intero più lungo a uno più breve, o un numero in virgola mobile a un intero, possono indurre il compilatore a lanciare un avvertimento, ma non sono illegali.

Il tipo `char` è semplicemente un intero piccolo, e lo si può usare liberamente nelle espressioni aritmetiche. Ciò consente notevole flessibilità nel manipolare certi tipi di trasformazioni di caratteri. Il lettore può trarre esempio da questa implementazione ingenua della funzione `atoi`, che converte una stringa di cifre nel suo equivalente numerico.

```
/* atoi: converte s in intero */
int atoi(char s[])
{
    int i, n;
    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

Come visto a suo tempo nel Capitolo 1, l'espressione

```
s[i] - '0'
```

fornisce il valore numerico del carattere contenuto in `s[i]`, poiché i valori di '`0`', '`1`' e così via formano una successione crescente di interi consecutivi.

Un altro esempio di conversione da `char` a `int` è dato dalla funzione `lower`, che serve a portare un carattere in minuscola *nel set di caratteri ASCII*. Se il carattere non è una lettera maiuscola, `lower` lo lascia immutato.

```
/* lower: converte c in minuscola; solo per tabella ASCII */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

Ciò funziona con la codifica ASCII poiché la differenza tra i valori numerici di una minuscola e della corrispondente maiuscola è una costante, ed entrambe le versioni dell'alfabeto (minuscola e maiuscola) non hanno soluzione di continuità: nella tabella ASCII non vi sono che lettere tra A e Z. Quest'ultima osservazione non vale per il set di caratteri EBCDIC, tuttavia: la nostra funzione convertirebbe qualcosa in più delle lettere, se usata con questa codifica.

L'intestazione standard `<ctype.h>`, descritta nell'Appendice B, definisce un gruppo di funzioni che forniscono test e conversioni non legate al set dei caratteri. Per esempio, `tolower(c)` restituisce la minuscola di `c` se `c` è in lettera maiuscola, sicché `tolower` è un'alternativa portabile alla funzione `lower` precedente.

Analogamente, la condizione

```
c >= '0' && c <= '9'
```

può essere sostituita da

```
isdigit(c)
```

D'ora in avanti faremo uso delle funzioni dell'intestazione `<ctype.h>`.

C'è un passaggio insidioso nella trasformazione dei caratteri in numeri interi. Il linguaggio non specifica se le variabili di tipo `char` siano quantità con o senza segno. Una conversione da `char` a `int` può risultare in un intero negativo? La risposta rispecchia le differenze di architettura, e dunque varia da una macchina all'altra. Su alcune macchine un `char` il cui bit più a sinistra sia 1 subirà la trasformazione in intero negativo ("estensione del segno"). Su altre, la promozione da `char` a `int` prevede semplicemente l'aggiunta di bit nulli a sinistra, con risultato sempre positivo.

La definizione del C assicura che i caratteri presenti nel set standard di caratteri stampabili della macchina non divengano mai negativi; ma configurazioni arbitrarie di bit memorizzate in variabili carattere possono risultare negative su alcune macchine, e positive su altre. Per garantire compatibilità, si usino gli attributi `signed` e `unsigned` quando occorre memorizzare valori arbitrari in variabili di tipo carattere.

Le espressioni relazionali come `i > j` e le espressioni logiche collegate con `&&` e `||` hanno valore 1 se vere, e zero se false. Ne consegue che l'assegnamento

```
d = c >= '0' && c <= '9'
```

imposta `d` a 1 se `c` è una cifra, e a 0 altrimenti. Tuttavia, funzioni come `isdigit` possono restituire qualunque valore diverso da zero per denotare il vero. La cosa non fa differenza, poiché nella porzione di `if`, `while`, `for` e così via in cui viene esaminata una condizione, "vero" significa solo "diverso da zero".

Le conversioni aritmetiche implicite non riservano grandi sorprese. Di norma, se un operatore come `+` o `*` con due operandi (un operatore binario) ha operandi di tipo diverso, il tipo "inferiore" è *promosso* al "superiore" prima che l'operazione proceda, e anche il risultato è del tipo superiore. Il Paragrafo 6 dell'Appendice A codifica con precisione le regole per la conversione. In assenza di operandi privi di segno, comunque, sarà sufficiente attenersi alle seguenti regole di massima:

Se uno degli operandi è `long double`, l'altro si converte in `long double`.

Altrimenti, se uno degli operandi è `double`, l'altro si converte in `double`.

Altrimenti, se uno degli operandi è `float`, l'altro si converte in `float`.

Altrimenti, `char` e `short` si convertono in `int`.

Quindi, se uno degli operandi è `long`, l'altro si converte in `long`.

Si noti che gli operandi di tipo `float` di un'espressione non sono trasformati automaticamente in `double`; questa è una modifica rispetto alla definizione originale del linguaggio. In genere, le funzioni matematiche come quelle contenute in `<math.h>` useranno la doppia precisione. Il motivo principale per cui si usa `float` è risparmiare memoria in vettori molto

grossi o, più raramente, risparmiare tempo in macchine in cui l'aritmetica a doppia precisione sia molto dispendiosa.

Le regole di conversione si complicano quando entrano in gioco operandi privi di segno: il problema sta nel fatto che i valori con e senza segno sono legati alla macchina, perché dipendono dalle dimensioni dei vari tipi di interi. Si supponga, per esempio, che `int` sia di 16 bit e `long` di 32 bit. Allora  $-1L < 1U$ , perché `1U`, che è un `unsigned int`, è promosso a `signed long`; ma  $-1L > 1UL$ , perché `-1L` è promosso a `unsigned long` e questo lo rende un numero positivo grande.

Gli assegnamenti implicano conversioni: il valore del membro destro è convertito nel tipo del membro sinistro, che è anche il tipo del risultato.

Un carattere è convertito in un intero, con o senza estensione del segno, come appena illustrato.

Gli interi più lunghi sono convertiti in interi più brevi (o in `char`) eliminando i bit più significativi in eccesso (cioè, da sinistra). Pertanto in

```
int i;
char c;

i = c;
c = i;
```

il valore di `c` resta immutato, a prescindere da qualunque considerazione sull'estensione del segno. Invertire l'ordine degli assegnamenti potrebbe, invece, far perdere informazioni.

Se `x` è `float` e `i` è `int`, allora le istruzioni `x = i` e `i = x` causano entrambe conversioni; la conversione da `float` a `int` provoca il troncamento della parte frazionaria. Quando `double` è convertito in `float`, possono verificarsi arrotondamenti o troncamenti del valore, a seconda dell'implementazione.

Visto che l'argomento di una chiamata di funzione è un'espressione, si assiste a conversioni di tipo anche nel passare argomenti alle funzioni. In mancanza di un prototipo di funzione, `char` e `short` diventano `int`, e `float` diventa `double`, il che spiega perché abbiamo dichiarato i parametri della funzione come `int` e `double` persino quando la funzione è chiamata con argomenti `char` e `float`.

Infine, è possibile imporre (o *forzare*) conversioni esplicite di tipo in qualsiasi espressione grazie a un operatore unario detto `cast`. Nel costrutto

`(nome-tipo) espressione`

`espressione` è convertita nel tipo desiderato secondo le regole di conversione sopra citate. Per capire il significato preciso del costrutto, si pensi a un assegnamento di `espressione` a una variabile del tipo specificato; il risultato dell'assegnamento è poi usato al posto dell'intero costrutto. Per esempio, la routine della libreria `sqrt` per il calcolo della radice quadrata si aspetta un argomento `double`, per cui genera una risposta assurda se erroneamente le viene passato qualcosa di diverso (`sqrt` è dichiarata nell'intestazione `<math.h>`). Se dunque `n` è un intero, si potrà usare

```
sqrt((double) n)
```

per convertire il valore di `n` in `double` prima di passarlo a `sqrt`. È bene notare che il costrutto `cast` genera il *valore* di `n` nel tipo appropriato; `n` stessa non è alterata. L'operatore `cast` ha il medesimo diritto di precedenza degli altri operatori unari, come mostrato nella Tabella 2.1.

Qualora gli argomenti siano dichiarati da un prototipo di funzione, come dovrebbero, la dichiarazione forza automaticamente la conversione di tutti gli argomenti quando si chiama la funzione. Pertanto, dato un prototipo di funzione per `sqrt`:

```
double sqrt(double);
```

la chiamata

```
root2 = sqrt(2);
```

forza l'intero 2 ad assumere il valore `double 2.0` senza alcun bisogno di `cast`.

La libreria standard comprende un'implementazione portabile di un generatore di numeri pseudo-casuali, `rand()`, e una funzione per impostarne il seme (*seed*); la prima funzione illustra un `cast`:

```
unsigned long int next = 1;

/* rand: restituisce un intero pseudo-casuale fra 0 e 32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: imposta il seme per rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

**Esercizio 2.3** Si scriva la funzione `htoi(s)` per convertire una stringa di cifre esadecimale (incluso un facoltativo `0x` o `0X`) nel valore intero corrispondente. Le cifre consentite sono quelle da `0` a `9`, da `a` a `f` e da `A` a `F`.

## 2.8 Operatori di incremento e decremento

Il C fornisce due operatori inusuali per l'incremento e il decremento dei valori delle variabili: `++` aggiunge 1 al suo operando, mentre `--` sottrae 1. Abbiamo usato spesso `++` per incrementare le variabili, come in

```
if (c == '\n')
    ++nl;
```

L'aspetto inusuale di questi operatori sta nel fatto che sono utilizzabili sia come prefisso (prima della variabile, come in `++n`) che come suffisso (dopo la variabile: `n++`). In entrambi i casi l'effetto è di incrementare il valore di `n`. Tuttavia, l'istruzione `++n` esegue l'incremento

*prima* di usare il valore di `n`, mentre `n++` lo fa *dopo* l'impiego del valore. Ne consegue che, nei contesti in cui è utilizzato il valore, e non solo l'effetto dell'incremento, le due forme conducono a risultati complessivi diversi: se `n` è 5, l'istruzione

```
x = n++;
```

imposta il valore di `x` a 5, mentre

```
x = ++n;
```

lo imposta a 6. In entrambi i casi il valore di `n` diviene 6. Gli operandi `++` e `--` possono essere applicati solo a variabili: espressioni come `(i+j)++` non sono ammesse. In contesti in cui non è richiesto alcun valore, ma solo l'effetto dell'incremento, come in

```
if (c == '\n')
    nl++;
```

le forme prefissa e suffisso si equivalgono. In certe situazioni, però, è necessario usare l'una o l'altra. Si consideri la seguente funzione `squeeze(s, c)`, che elimina tutte le occorrenze del carattere `c` dalla stringa `s`.

```
/* squeeze: elimina tutte le occorrenze di c da s */
void squeeze(char s[], int c)
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

Ogni volta che si incontra un carattere diverso da `c`, esso viene copiato nella posizione corrente `j`, e solo allora si incrementa `j` pronto per il carattere successivo. Il costrutto `if` riportato sopra è in tutto equivalente a

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

Un altro esempio simile è dato dalla funzione `getline` del Capitolo 1, dove è possibile sostituire

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

con il frammento più compatto

```
if (c == '\n')
    s[i++] = c;
```

Come terzo esempio, si consideri la funzione standard `strcat(s,t)`, che concatena la stringa `t` ponendola in coda alla stringa `s`. (La funzione assume che `s` sia sufficientemente lunga da contenere la concatenazione.) Nella versione data di seguito, `strcat` non restituisce alcun valore, mentre la sua omonima della libreria standard restituisce un puntatore alla stringa risultante.

```
/* strcat: concatena t in coda a s; s deve essere sufficientemente lunga */
void strcat(char s[], char t[])
{
    int i, j;

    i = j = 0;
    while (s[i] != '\0') /* trova la fine di s */
        i++;
    while ((s[i++] = t[j++]) != '\0') /* copia t */
        ;
}
```

Nel copiare i caratteri da `t` a `s`, il suffisso `++` è applicato sia a `i` che a `j` per garantire che i loro valori indichino le posizioni corrette per l'iterazione successiva del ciclo `while`.

**Esercizio 2.4** Si scriva una variante di `squeeze(s1,s2)` che elimini tutte le occorrenze di caratteri di `s1` che appaiono anche nella stringa `s2`.

**Esercizio 2.5** Si scriva una funzione `any(s1,s2)` che restituisca la prima posizione nella stringa `s1` nella quale si trovi un'occorrenza di uno qualunque fra i caratteri di `s2`, o restituisca `-1` nel caso in cui nessun carattere di `s2` appaia anche in `s1`. (La funzione standard `strpbrk` esegue lo stesso compito, ma restituisce anche un puntatore alla posizione cercata.)

## 2.9 Operatori per la manipolazione dei bit

Il C prevede sei operatori per la manipolazione dei bit, applicabili solo a operandi interi, os-sia di tipo `char`, `short`, `int` e `long`, con o senza segno.

- & AND bit per bit
- | OR bit per bit
- ^ OR esclusivo bit per bit
- << scorrimento a sinistra
- >> scorrimento a destra
- complemento a uno (unario)

L'operatore `&` è spesso usato per azzerare un insieme di bit; per esempio,

```
n = n & 0177;
```

imposta a zero tutti i bit di `n`, eccetto i 7 meno significativi (quelli più a destra).

L'operatore `|` si usa invece per impostare a 1 certi bit. L'istruzione

```
x = x | SET_ON;
```

pone a uno tutti i bit di `x` che corrispondono a bit di `SET_ON` di valore 1.

Costanti come `SET_ON` e `0177`, impiegate per impostare i valori dei bit di un certo campo, sono dette *maschere* ("masks").

L'operatore OR esclusivo bit per bit `^` imposta un 1 in ciascuna posizione in cui gli operandi presentano bit diversi, e zero ove essi siano uguali.

Bisogna evitare di confondere gli operatori AND e OR bit per bit (`&` e `|`) con gli operatori logici di congiunzione e disgiunzione (`&&` e `||`), che seguono l'ordine di valutazione da sinistra a destra. Per esempio, se `x` vale 1 e `y` è 2, allora `x & y` è zero, ma `x && y` è uno.

Gli operatori di scorrimento `<<` e `>>` eseguono scorrimenti a sinistra o a destra del loro operando sinistro in base al numero di posizioni indicato dall'operando destro, che deve essere non negativo. Quindi, `x << 2` fa scorrere i bit di `x` a sinistra di due posizioni, sostituendo i due bit più a destra di `x` con dei nuovi bit nulli; ciò equivale a moltiplicare `x` per 4. Lo scorrimento a destra di una quantità di tipo `unsigned` rimpiazza sempre i bit persi con degli zeri; se la quantità è però con segno, alcune macchine sostituiranno i bit persi con bit di segno ("scorrimento aritmetico"), mentre altre con bit nulli ("scorrimento logico").

L'operatore unario `-` produce il complemento a uno dell'operando; ossia, converte ogni bit dell'operando nella sua negazione: 1 diviene 0, e 0 diviene 1. Per esempio,

```
x = x & ~077
```

imposta gli ultimi sei bit di `x` a zero. Si noti che l'espressione `x & ~077` è indipendente dalla dimensione di `x`, ed è quindi da preferirsi, per esempio, a `x & 01777700`, che presuppone che `x` sia di 16 bit. La forma portabile non è meno economica, perché `~077` è un'espressione costante che può essere valutata durante la compilazione.

Per illustrare alcuni degli operatori in oggetto, si consideri una funzione `getbits(x,p,n)` che restituisca `n` bit consecutivi di `x` a partire dalla posizione `p`, allineando il risultato a destra. Assumiamo che la posizione 0 denoti il bit meno significativo di `x` (il primo da destra), e che `n` e `p` siano valori sensati. Quindi, `getbits(x,4,3)` restituisce i tre bit di `x` in posizione 4, 3 e 2, allineati a destra.

```
/* getbits: estrae n bit di x a partire dalla posizione p */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(-0 << n);
}
```

L'espressione `x >> (p+1-n)` sposta la successione desiderata di bit all'estremo destro della parola. La costante `-0` è costituita esclusivamente da bit posti a 1, e il suo scorrimento a sinistra di `n` posizioni `-0<<n` imposta a zero `n` bit a partire da destra; il complemento a uno del risultato genera una maschera i cui primi `n` bit da destra sono posti a uno, e gli altri a zero.

**Esercizio 2.6** Si scriva una funzione `setbits(x,p,n,y)` che restituisca `x` con gli `n` bit a partire dalla posizione `p` uguali agli `n` bit meno significativi di `y`, lasciando immutati i rimanenti bit di `x`.

**Esercizio 2.7** Si scriva una funzione `invert(x,p,n)` che restituisca `x` con gli `n` bit a partire dalla posizione `p` negati, e gli altri immutati.

**Esercizio 2.8** Si scriva una funzione `rightrot(x, n)` che restituisca il valore dell'intero `x` dopo che i suoi bit sono stati ruotati a destra di `n` posizioni. (Una rotazione a destra differisce da uno scorrimento a destra in quanto i bit che “fuoriescono” da destra non vanno persi, ma “rientrano” da sinistra.)

## 2.10 Operatori di assegnamento ed espressioni

Espressioni come

`i = i + 2`

nelle quali la variabile a sinistra è immediatamente ripetuta nel membro destro, si possono sinteticamente riscrivere nella forma

`i += 2`

L'operatore `+=` è detto *operatore di assegnamento*.

Molti degli operatori binari, ovvero che agiscono su due operandi (come l'addizione `+`) hanno un corrispondente operatore di assegnamento `op=`, dove `op` può essere

`+ - * / % << >> & ^ |`

Partendo dalle espressioni `expr1` e `expr2`,

`expr1 op= expr2`

equivale a

`expr1 = (expr1) op (expr2)`

salvo per il fatto che, nella prima forma, `expr1` è computata una sola volta. Si notino le parentesi che racchiudono `expr2`:

`x *= y + 1`

sta per

`x = x * (y + 1)`

e non per

`x = x * y + 1`

Come esempio, la funzione `bitcount` conta il numero di bit pari a 1 del suo argomento intero.

```
/* bitcount: conta i bit pari a 1 di x */
int bitcount(unsigned x)
{
    int b;

    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            b++;
    return b;
}
```

La dichiarazione di `x` come `unsigned` assicura che, dopo lo scorrimento a destra, il valore dei nuovi bit entranti da sinistra sia proprio 0, e non invece un possibile bit di segno, indipendentemente dalla macchina su cui il programma risiede.

Al di là della loro brevità, gli operatori di assegnamento risultano naturali perché vicini al modo di pensare di un essere umano. Si dice “aggiungi 2 a `i`”, oppure “incrementa `i` di 2 unità”, e non “computa la somma fra 2 e `i`, e memorizza il risultato in `i`”. È per questo che l'espressione `i += 2` è da preferirsi a `i = i+2`. Inoltre, per espressioni complicate come

`yyval[yypv[p3+p4] + yypv[p1+p2]] += 2`

l'operatore di assegnamento rende il codice più trasparente, perché il lettore non ha bisogno di controllare se due involute espressioni siano coincidenti, o di chiedersi perché mai non lo siano. Inoltre, un operatore di assegnamento può favorire la produzione di codice più efficiente da parte del compilatore.

Si è già visto come le istruzioni di assegnamento abbiano un valore riutilizzabile nelle espressioni; l'esempio principe è

```
while ((c = getchar()) != EOF)
    ...

```

Anche gli altri operatori di assegnamento (`+=`, `-=` e così via) possono comparire nelle espressioni, sebbene la cosa sia più rara.

Il tipo di un'espressione di assegnamento è quello del suo operando sinistro, e il suo valore è il valore di detto operando dopo l'esecuzione dell'assegnamento.

**Esercizio 2.9** In un sistema di numerazione con complemento a due, `x &= (x-1)` pone a zero il primo bit di `x` pari a 1 a partire da destra. Si spieghi perché. Si sfrutti questa osservazione per scrivere una versione più veloce di `bitcount`.

## 2.11 Espressioni condizionali

Le istruzioni

```
if (a > b)
    z = a;
else
    z = b;
```

assegnano a `z` il massimo valore fra `a` e `b`. Le *espressioni condizionali*, che sfruttano l'operatore ternario “`? :`”, permettono di parafrasare simili brani di codice. Nella forma

`expr1 ? expr2 : expr3`

l'espressione `expr1` è valutata per prima. Se il suo valore non è zero (cioè, se è vera), è valutata `expr2`, e il risultato è il valore dell'espressione condizionale; altrimenti viene valutata `expr3`, il cui valore diviene il valore dell'espressione condizionale. In ogni caso, si valuta una sola fra le espressioni `expr2` e `expr3`. Per impostare `z` al massimo tra `a` e `b` basta quindi scrivere

`z = (a > b) ? a : b; /* z = max(a, b) */`

Si noti che l'espressione condizionale è una vera e propria espressione, il cui uso è quindi ammesso ognqualvolta sia ammessa un'espressione. Se  $espr_2$  e  $espr_3$  sono di tipo diverso, il tipo dell'espressione condizionale è determinato dalle regole di conversione già discusse in questo capitolo. Per esempio, se  $f$  è di tipo `float` e  $n$  è di tipo `int`, l'espressione

```
(n > 0) ? f : n
```

è di tipo `float`, indipendentemente dalla positività di  $n$ .

Non è necessario racchiudere tra parentesi la prima espressione di un'espressione condizionale, perché il grado di precedenza di `:` è molto modesto, appena superiore a quello dell'assegnamento. Si consiglia comunque di usare le parentesi per migliorare la leggibilità.

Le espressioni condizionali sono spesso foriere di codice molto sintetico. Per fare un esempio, il ciclo seguente visualizza  $n$  elementi di un vettore, 10 per riga, su colonne separate da uno spazio; ogni riga, inclusa l'ultima, termina con un carattere newline.

```
for (i = 0; i < n; i++)
    printf("%d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' '');
```

Un carattere newline è inserito ogni dieci elementi del vettore, e dopo l' $n$ -esimo, mentre tutti gli altri elementi sono seguiti da uno spazio. Il codice può sembrare involuto, ma è più compatto dell'equivalente basato sul costrutto `if-else`. Un altro buon esempio è

```
printf("Hai %d oggetti\n", n, n==1 ? "o" : "i");
```

**Esercizio 2.10** Si riscriva la funzione `lower`, che converte lettere maiuscole in minuscole, sfruttando un'espressione condizionale in luogo del costrutto `if-else`.

## 2.12 Precedenze e ordine di valutazione

La Tabella 2.1 riassume i diritti di precedenza e l'associatività di tutti gli operatori, inclusi quelli non ancora esaminati. Gli operatori su una stessa riga hanno la stessa precedenza, e hanno diritto di precedenza sugli operatori alle righe successive. Così, per esempio, `*`, `/` e `%` hanno pari precedenza, superiore a quella degli operatori binari `+` e `-`. L'“operatore” `()` si riferisce alle chiamate di funzione, mentre `->` e `.` danno accesso ai membri delle strutture, e saranno presentati nel Capitolo 6 assieme a `sizeof`, che fornisce la dimensione di un oggetto. Il Capitolo 5 tratta dell'indirezione dei puntatori `(*)` e dell'indirizzo degli oggetti `(&)`, e il Capitolo 3 si occupa dell'operatore virgola.

Si noti che `==` e `!=` hanno diritto di precedenza su `&`, `^` e `!`, il che implica la necessità di inserire le parentesi in espressioni che confrontino quantità bit per bit, come

```
if ((x & MASK) == 0) ...
```

pena un comportamento indesiderato del programma.

Come molti altri linguaggi, anche il C non specifica l'ordine nel quale si valutano gli operandi di un operatore (fanno eccezione a questa regola `||`, `&&`, `?:` e `,`). Per esempio, nell'espressione

```
x = f() + g();
```

**Tabella 2.1** Associatività e diritto di precedenza fra gli operatori.

OPERATORI	ASSOCIAZIVITÀ
<code>() [] -&gt; .</code>	da sinistra a destra
<code>! ~ ++ -- + - * &amp; (tipo) sizeof</code>	da destra a sinistra
<code>* / %</code>	da sinistra a destra
<code>* -</code>	da sinistra a destra
<code>&lt;&lt; &gt;&gt;</code>	da sinistra a destra
<code>&lt; &lt;= &gt; &gt;=</code>	da sinistra a destra
<code>== !=</code>	da sinistra a destra
<code>&amp;</code>	da sinistra a destra
<code>^</code>	da sinistra a destra
<code>!</code>	da sinistra a destra
<code>&amp;&amp;</code>	da sinistra a destra
<code>  </code>	da sinistra a destra
<code>?:</code>	da destra a sinistra
<code>= += -= *= /= %= &amp;= ^=  = &lt;&lt;= &gt;&gt;=</code>	da destra a sinistra
<code>,</code>	da sinistra a destra

Gli operatori unari `+`, `-`, `&` e `*` hanno diritto di precedenza sui loro omonimi binari.

può accadere che sia valutata prima  $f$  o prima  $g$ , e se una delle due funzioni altera il valore di una variabile dalla quale l'altra dipende, lo stesso valore di  $x$  può risultare dipendente dall'ordine di valutazione. Per evitare questo inconveniente, è sufficiente memorizzare i risultati delle valutazioni parziali in variabili ausiliarie, nell'ordine in cui si intende far valutare l'espressione.

Analogamente, non è dato sapere l'ordine di valutazione degli argomenti di una funzione, per cui l'istruzione

```
printf("%d %d\n", ++n, power(2, n)); /* CODICE ERRATO */
```

può portare a risultati diversi con compilatori diversi, a seconda del fatto che  $n$  sia o meno incrementato prima della chiamata di `power`. La soluzione, naturalmente, è questa:

```
++n;
printf("%d %d\n", n, power(2,n));
```

Le chiamate di funzione, le istruzioni di assegnamento annidate e gli operatori di incremento e decremento provocano tali effetti collaterali: qualche variabile cambia valore a seguito della valutazione dell'espressione. I valori delle espressioni possono dipendere alquanto sottilmente da tali effetti collaterali – dall'ordine, cioè, in cui le variabili coinvolte sono aggiornate.

Una circostanza davvero infelice è rappresentata dall'istruzione

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

Il punto cruciale, qui, è se l'indice del vettore sia il vecchio o il nuovo valore di *i*. La risposta dipende dal compilatore: è ben possibile ottenere comportamenti diversi con compilatori diversi. Lo standard, in questo e molti altri casi simili, lascia volutamente libertà di scelta, perché il miglior ordine di valutazione delle espressioni è strettamente correlato all'architettura della macchina. Ne consegue che la presenza di effetti collaterali indesiderati (assegnamenti fra variabili) dipende dal compilatore. (Lo standard, tuttavia, specifica almeno che tutti gli eventuali effetti collaterali nella valutazione degli argomenti di una funzione abbiano luogo prima della chiamata della funzione stessa: il che però non aiuta nell'esempio relativo a `printf` appena illustrato.)

La morale che se ne può trarre è la seguente: in qualunque linguaggio, scrivere codice che sia dipendente dall'ordine di valutazione delle espressioni è cattiva programmazione. Vada sé che è necessario sapere cosa evitare; ignorare del tutto lo specifico ordine di valutazione adottato dai vari sistemi aiuta a non cadere nella tentazione di sfruttare le peculiarità di una data implementazione del linguaggio.

# 3

## Flusso del controllo

**L**E ISTRUZIONI PER IL FLUSSO DEL CONTROLLO di un linguaggio prescrivono l'ordine seguito dalla computazione. Abbiamo già incontrato i costrutti più comuni di flusso del controllo negli esempi precedenti; qui tratteremo i rimanenti, approfondendo anche quelli già analizzati.

### 3.1 Istruzioni e blocchi

Un'espressione come `x = 0 o i++ o printf(...)` diventa un'*istruzione* quando è seguita da un punto e virgola, come in

```
x = 0;
i++;
printf(...);
```

Il punto e virgola, anziché fungere da separatore, come per esempio nel Pascal, nel C serve a terminare l'istruzione.

Le parentesi graffe sono usate per raggruppare dichiarazioni e istruzioni, il che genera un'*istruzione composta*, o *blocco*; sono equivalenti, sul piano della sintassi, a un'istruzione singola. Le parentesi graffe che circondano le istruzioni di una funzione sono un esempio ovvio; un altro sono le graffe attorno a istruzioni multiple dopo un costrutto `if`, `else`, `while` o `for`. (Le variabili possono essere dichiarate in *qualsiasi* blocco; riprenderemo l'argomento nel Capitolo 4.) Alla parentesi graffa di chiusura di un blocco non segue il punto e virgola.

## 3.2 If-else

L'istruzione `if-else` serve per prendere decisioni. Formalmente, la sintassi è

```
if (espressione)
    istruzione1
else
    istruzione2
```

dove la clausola `else` è facoltativa. La `espressione` è valutata; se è vera (se cioè `espressione` ha un valore diverso da zero), è eseguita `istruzione1`. Se è falsa (`espressione` è zero) e se c'è una clausola `else`, viene invece eseguita `istruzione2`.

Poiché l'istruzione `if` si limita a verificare il valore numerico di un'espressione, è possibile ricorrere ad alcune scorciatoie nella codifica. La più ovvia consiste nello scrivere

```
if (espressione)
```

in sostituzione di

```
if (espressione != 0)
```

Talvolta questa forma è chiara e naturale; in altri casi può risultare oscura.

Stante la natura facoltativa della clausola `else` in un costrutto `if-else`, si crea ambiguità quando un `else` è omesso da una sequenza `if` annidata. Per convenzione, il compilatore abbinia `else` al più vicino `if` precedente che ne sia privo. Per esempio in

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

la clausola `else` va con l'istruzione `if` più interna, come segnala la rientranza. Se si vuole un risultato diverso, è necessario ricorrere alle parentesi graffe per impostare il giusto abbinamento:

```
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
```

L'ambiguità è particolarmente pericolosa in situazioni come la seguente:

```
if (n >= 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf(...);
            return i;
        }
else /* ERRATO */
    printf("errore: n e' negativo\n");
```

Il rientro evidenzia chiaramente quale risultato ci si aspetta, ma il compilatore non ne tiene conto e abbina la clausola `else` all'istruzione `if` più interna. Questo tipo di errore può essere difficile da scoprire; è una buona idea usare le parentesi graffe con le istruzioni `if` annidate.

Per inciso, si noti la presenza del punto e virgola dopo `z = a` in

```
if (a > b)
    z = a;
else
    z = b;
```

Questo perché, secondo la grammatica C, il costrutto `if` richiede in quel punto un'istruzione, e un'espressione come “`z = a`” diventa istruzione solo se seguita da un punto e virgola.

## 3.3 Else-if

Il costrutto

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

ricorre così spesso da meritare un breve excursus. Questa sequenza di istruzioni `if` è il modo più generale di analizzare un ventaglio di possibilità in C. Le `espressioni` sono valutate nell'ordine; se una di esse è vera, l'`istruzione` con cui è associata viene eseguita, e ciò conclude l'esecuzione dell'intero costrutto. Come sempre, il codice per ogni `istruzione` può essere un'istruzione singola o un blocco (più istruzioni tra graffe).

L'ultima clausola `else` si occupa del caso “nessuno dei precedenti” in cui, non essendo soddisfatta nessuna delle condizioni, occorre procedere d'ufficio. Può succedere che nessuna azione esplicita sia prevista per questo caso, e allora il passo

```
else
    istruzione
```

può essere tralasciato, oppure lo si può riservare alla gestione degli errori, per rilevare una condizione “impossibile”.

Per illustrare una scelta fra tre rami, prendiamo in esame una funzione di ricerca binaria che decide se un particolare valore `x` si trova nel vettore ordinato `v`. Gli elementi di `v` devono disporsi in ordine crescente. La funzione restituisce la posizione (un numero fra 0 e `n-1`) se `x` compare in `v`, e il valore -1 in caso contrario.

La ricerca binaria confronta in primo luogo il valore  $x$  in ingresso con l'elemento mediano del vettore  $v$ . Se  $x$  è il minore fra i due, la ricerca si concentra sulla metà inferiore del vettore, in caso contrario sulla metà superiore. In entrambi i casi, la fase successiva prevede il confronto di  $x$  con l'elemento mediano della metà interessata. Il processo di divisione in due dell'intervallo continua finché si è trovato il valore cercato, o si sono esaminati tutti i casi possibili.

```
/* binsearch: trova x in v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else          /* trovato un elemento coincidente con x */
            return mid;
    }
    return -1;      /* x non compare nel vettore v */
}
```

Il nodo fondamentale è, a ogni passo, decidere se  $x$  sia minore, maggiore o uguale all'elemento mediano  $v[mid]$ ; questo è un compito naturale del costrutto `else-if`.

**Esercizio 3.1** La nostra implementazione della ricerca binaria esamina due condizioni all'interno del ciclo, mentre ne basterebbe una (a costo, però, di altre condizioni da esaminare all'esterno). Si scriva una versione con una sola condizione interna del ciclo, misurando la differenza nei tempi di esecuzione.

## 3.4 Switch

L'istruzione `switch` esamina un ventaglio finito di possibilità, e devia di conseguenza il flusso dell'esecuzione. Ogni condizione esaminata esprime la coincidenza del valore di un'espressione con una costante intera.

```
switch (espressione) {
    case espressione-costante: istruzioni
    case espressione-costante: istruzioni
    default:                  istruzioni
}
```

Ogni caso (clausola `case`) è etichettato da una o più costanti (o espressioni costanti) intere. Se uno di tali casi coincide con il valore dell'espressione, l'esecuzione prosegue dal caso in

questione. Tutte le espressioni delle clausole `case` devono essere diverse. La clausola `default` è eseguita se nessuna delle altre è soddisfatta, ed è facoltativa; in sua assenza, se gli altri casi non sono soddisfatti non accade nulla: l'istruzione `switch` non ha alcun effetto. I casi e la clausola `default` possono trovarsi in qualunque ordine.

Nel Capitolo 1 abbiamo scritto un programma per contare cifre, spazi e altri caratteri in un testo attraverso una sequenza `if... else if... else`. Ecco ora lo stesso programma con il costrutto `switch`:

```
#include <stdio.h>

main() /* conta cifre, spazi e caratteri rimanenti del testo in ingresso */
{
    int c, i, nwhite, nother, ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
        ndigit[i] = 0;
    while ((c = getchar()) != EOF) {
        switch (c) {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8': case '9':
                ndigit[c-'0']++;
                break;
            case ' ':
            case '\n':
            case '\t':
                nwhite++;
                break;
            default:
                nother++;
                break;
        }
    }
    printf("cifre =");
    for (i = 0; i < 10; i++)
        printf(" %d", ndigit[i]);
    printf(", spazi = %d, altri = %d\n",
           nwhite, nother);
    return 0;
}
```

L'istruzione `break` provoca l'uscita immediata dal costrutto `switch`. Poiché i casi fungono solo da etichette, quando l'esecuzione del codice relativo a un caso è finita, essa passa al successivo, a meno che non si decida esplicitamente di fare altrimenti. Questa gestione del flusso è detta "a cascata" (*falling through*). Le istruzioni `break` e `return` sono le più comuni vie d'uscita da `switch`. Un'istruzione `break` impone anche l'uscita immediata dai cicli `while`, `for` e `do`, come si vedrà nel seguito di questo capitolo.

L'esecuzione a cascata presenta vantaggi e svantaggi. Il lato positivo è la possibilità di associare molti casi a una singola azione, come avviene per le diverse cifre nell'esempio, ma implica anche, di norma, che ogni caso termini con un `break` per evitare il passaggio in cascata al successivo: tale meccanismo, infatti, non è robusto ed espone il brano `switch` a una possibile disintegrazione in caso di modifica del programma. Fatto salvo l'uso di più etichette in una singola computazione, l'esecuzione a cascata va usata con parsimonia e corredato con appositi commenti.

A beneficio della forma, si inserisce sempre un'istruzione `break` dopo l'ultimo caso (che qui è `default`), anche se la logica non lo richiederebbe. Se in un futuro un altro caso dovesse essere aggiunto alla fine del costrutto, questo stratagemma eviterà catastrofi.

**Esercizio 3.2** Si scriva una funzione `escape(s,t)` che, nel copiare la stringa `t` in `s`, trasformi i caratteri newline e le tabulazioni in sequenze di controllo visibili quali `\t` e `\n`. Si adoperi l'istruzione `switch`. Si scriva anche la funzione inversa, così da convertire le sequenze di controllo nei veri caratteri corrispondenti.

### 3.5 Cicli: while e for

Abbiamo già fatto conoscenza con i cicli `while` e `for`. In

```
while (espressione)
    istruzione
```

*espressione* è valutata. Qualora sia diversa da zero, *istruzione* è eseguita ed *espressione* rivalutata. Questo ciclo prosegue finché *espressione* diventa zero, momento in cui l'esecuzione riprende dal punto seguente a *istruzione*.

L'istruzione `for`

```
for (espressione1; espressione2; espressione3)
    istruzione
```

è equivalente a

```
espressione1;
while (espressione2) {
    istruzione
    espressione3;
}
```

a eccezione del comportamento di `continue`, descritto nel Paragrafo 3.7.

Sul piano grammaticale, le tre componenti di un ciclo `for` sono espressioni. Nel caso più comune, *espressione*<sub>1</sub> ed *espressione*<sub>3</sub> sono assegnamenti o chiamate di funzioni, mentre *espressione*<sub>2</sub> è un'espressione relazionale. Si può omettere una delle tre, ma i punti e virgola devono rimanere. Omettere *espressione*<sub>1</sub> o *espressione*<sub>3</sub> nell'istruzione `for` corrisponde a ometterle nell'equivalente ciclo `while`. Se però la condizione, cioè *espressione*<sub>2</sub>, non è presente, essa è considerata sempre vera, dunque

```
for (;;) {
    ...
}
```

è un ciclo "infinito", presumibilmente da interrompere, prima o poi, con altri strumenti come `break` o `return`.

Se sia meglio usare `while` o `for` è sostanzialmente una questione di preferenza personale. Per esempio, in

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ; /* salta gli spazi */
```

non si ha inizializzazione o reinizializzazione, quindi è più naturale usare `while`.

L'istruzione `for` è preferibile in presenza di inizializzazione e incremento semplici, dato che raggruppa le istruzioni di controllo del ciclo in posizione ben visibile all'inizio dello stesso. Questo risulta palesemente da

```
for (i = 0; i < n; i++)
    ...
```

che è proprio del linguaggio C per elaborare i primi `n` elementi di un vettore, in analogia al ciclo DO del Fortran o for del Pascal. È tuttavia un'analogia imperfetta, visto che l'indice e il limite di un ciclo `for` nel C possono essere alterati dall'interno del ciclo, e la variabile indice `i` mantiene il suo valore quando il ciclo, per qualunque ragione, termina. Poiché `for` è composto da espressioni arbitrarie, i cicli `for` non sono vincolati alle espressioni aritmetiche; ma ciò non toglie che sia poco elegante forzare computazioni tra loro scollegate all'interno delle porzioni di inizializzazione e incremento di un `for`, che è più opportuno riservare alle operazioni di controllo del ciclo.

Come esempio di più vasta portata, esaminiamo un'altra versione di `atoi` per convertire una stringa nel suo equivalente numerico. Questa versione è leggermente più generale di quella vista nel Capitolo 2; considera anche eventuali spazi bianchi iniziali e un segno + o - facoltativo (il Capitolo 4 presenta `atof`, che opera la stessa conversione per i numeri con virgola mobile).

La struttura del programma riflette la forma dei dati in ingresso:

```
salta gli spazi iniziali se ve ne sono
appura il segno, se c'è
leggi i caratteri che rappresentano cifre, e convertili
```

Ogni fase fa il suo lavoro, lasciando le cose in ordine per la successiva. L'intero processo ha termine al primo carattere che non può fare parte di un numero.

```
#include <ctype.h>
/* atoi: converte s in intero; seconda versione */
int atoi(char s[])
{
    int i, n, sign;

    for (i = 0; isspace(s[i]); i++) /* salta gli spazi iniziali */
    ;
    sign = (s[i] == '-') ? -1 : 1; /* appura il segno */
    if (s[i] == '+' || s[i] == '-') /* se vi e' un segno, */
        i++;
    for (n = 0; isdigit(s[i]); i++) /* passa avanti di un carattere */
        ...
```

```

    n = 10 * n + (s[i] - '0');
    return sign * n;
}

```

La libreria standard fornisce `strtol`, una funzione più elaborata per la conversione delle stringhe in numeri interi lunghi; si veda il Paragrafo 5 dell'Appendice B.

I vantaggi di mantenere un controllo centralizzato sul ciclo diventano ancora più evidenti in presenza di cicli annidati. La seguente funzione codifica l'algoritmo di Shell, inventato nel 1959 da D. L. Shell e noto come `shellsort`, per ordinare un vettore di interi. L'idea chiave è che negli stadi iniziali dell'esecuzione si confrontano elementi distanti anziché contigui, come accade nei più semplici ordinamenti per interscambio. Questo algoritmo tende a eliminare rapidamente grandi quantità di disordine, il che facilita il lavoro successivo. L'intervallo tra gli elementi confrontati si riduce per gradi fino a uno, e da qui in poi l'ordinamento segue il metodo di interscambio tra elementi adiacenti. L'algoritmo è noto in forma semplificata come `shellsort`.

```

/* shellsort: ordina v[0]...v[n-1] in ordine crescente */
void shellsort(int v[], int n)
{
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}

```

I cicli annidati sono tre. Il più esterno controlla l'intervallo tra gli elementi confrontati, riducendolo da  $n/2$  di un fattore due a ogni iterazione, fino a zero. Il ciclo mediano scandisce gli elementi. Il ciclo più interno confronta ciascuna coppia di elementi separata da `gap` e inverse quelle fuori posto. Dato che `gap` si riduce a uno, tutti gli elementi verranno ordinati correttamente. Si noti come la generalità del costrutto `for` renda il ciclo più esterno formalmente omogeneo agli altri, sebbene l'indice `gap` non segua una progressione aritmetica.

Un altro operatore del C è la virgola “,”, quasi sempre usata nell'istruzione `for`. Una coppia di espressioni separate da una virgola è valutata da sinistra a destra, e il tipo e il valore del risultato sono quelli dell'operando a destra. Pertanto è possibile collocare espressioni multiple in varie porzioni dell'istruzione `for`, per esempio per analizzare due indici contemporaneamente, come illustra la funzione `reverse(s)`, che inverte `s` senza copiarla in una stringa ausiliaria.

```

#include <string.h>
/* reverse: inverte la stringa s senza eseguirne una copia */
void reverse(char s[])
{

```

```

    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}

```

Le virgolette che dividono gli argomenti della funzione, le variabili nelle dichiarazioni e così via *non* sono operatori virgola, e non garantiscono la valutazione da sinistra a destra.

Gli operatori virgola vanno usati con parsimonia. Gli usi più idonei sono in costrutti formalmente connessi l'uno con l'altro, come nel ciclo `for` in `reverse` e nelle macro, dove una computazione a più passi deve consistere in una singola espressione. Un operatore virgola potrebbe anche essere appropriato per lo scambio di elementi in `reverse`, dove tale scambio va concepito come operazione singola:

```

for (i = 0, j = strlen(s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;

```

**Esercizio 3.3** Si scriva una funzione `expand(s1,s2)` che espanda le notazioni stenografiche come `a-z` della stringa `s1` nel corrispondente elenco completo `abc...xyz` in `s2`. Il lettore provveda alle minuscole, alle maiuscole, alle cifre e si prepari ad affrontare casi come `a-b-c`, `a-z0-9` e `-a-z`. Si faccia in modo che un segno – in testa o in coda sia interpretato dal programma in senso letterale.

## 3.6 Cicli: do-while

Come abbiamo visto nel Capitolo 1, i cicli `while` e `for` valutano la condizione di terminazione fin dall'inizio. Il terzo ciclo del C, l'istruzione `do-while`, esamina invece la condizione in coda, dopo ogni iterazione: il corpo è quindi sempre eseguito almeno una volta.

La sintassi del costrutto è

```

do
    istruzione
    while (espressione);

```

Si esegue dapprima `istruzione`, e si valuta poi `espressione`: se è vera, si torna a eseguire `istruzione`, e così via. Il ciclo termina quando `espressione` diventa falsa. Il costrutto `do-while` è equivalente all'istruzione `repeat-until` del Pascal, ma in quest'ultimo caso l'iterazione termina se la condizione diventa vera.

L'esperienza dimostra che l'istruzione `do-while` è molto meno usata di `while` e `for`. Nondimeno a volte torna utile, come nel caso della funzione seguente, `itoa`, che converte un intero in una stringa, ed è quindi l'inversa di `atoi`. Il compito è un po' più complicato di quanto sembri a prima vista: i metodi più semplici per generare le cifre lo fanno nell'ordine sbagliato. La nostra soluzione è di generare l'immagine speculare della stringa corretta, per poi capovolgerla.

```

/* itoa: converte n nella stringa di caratteri s */
void itoa(int n, char s[])
{
    int i, sign;

    if ((sign = n) < 0) /* tiene traccia del segno */
        n = -n;           /* rende n positivo */
    i = 0;
    do {                 /* genera le cifre nell'ordine inverso */
        s[i++] = n % 10 + '0'; /* estrae la cifra seguente*/
    } while ((n /= 10) > 0); /* elimina la cifra da n */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}

```

Il ciclo `do-while` è necessario, o perlomeno conveniente, perché il vettore `s` dovrà sempre contenere almeno un carattere, anche se `n` fosse zero. Abbiamo racchiuso l'unica istruzione del corpo del ciclo tra graffe che, benché non necessarie, aiuteranno il lettore frettoloso a non scambiare l'istruzione `while` per l'*inizio* di un ciclo `while`.

**Esercizio 3.4** In un sistema con complemento a due, la nostra versione di `itoa` non gestisce correttamente il massimo numero negativo ammesso in ingresso, cioè il valore di `n` pari a  $-2^{\text{dimensione parola}-1}$ . Si spieghi perché, e si apportino le modifiche necessarie per visualizzare tale valore correttamente, indipendentemente dal tipo di macchina.

**Esercizio 3.5** Si scriva una funzione `itob(n,s,b)` che converta l'intero `n` in una stringa `s` i cui caratteri rappresentano le cifre di `n` in base `b`. In particolare, `itob(n,s,16)` converte `n` in un intero esadecimale rappresentato in `s`.

**Esercizio 3.6** Si stenda una versione di `itoa` che accetti tre argomenti invece di due, il terzo dei quali denoti l'ampiezza minima del campo. Il numero convertito deve essere integrato da spazi a sinistra fino a raggiungere l'ampiezza sufficiente.

### 3.7 Break e continue

Torna a volte conveniente interrompere un'iterazione in corrispondenza di punti del codice diversi dall'inizio o dalla fine del corpo. L'istruzione `break` serve allo scopo nel caso di istruzioni `for`, `while` e `do`, in analogia con il suo uso nel costrutto `switch`. Essa provoca l'immediata terminazione del ciclo o dell'istruzione `switch`.

La funzione `trim` rimuove spazi, tabulazioni e caratteri newline in coda a una stringa, sfruttando `break` per uscire da un ciclo quando incontra il primo carattere da destra diverso da quelli indicati.

```

/* trim: rimuove spazi, tab e caratteri newline in coda a s */
int trim(char s[])
{
    int n;

```

```

    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}

```

La funzione `strlen` della libreria restituisce la lunghezza della stringa. Il ciclo `for` itera a ritroso dalla coda della stringa, cercando il primo carattere diverso da ' ', '\t' e '\n'. Il ciclo termina quando si incontra uno dei caratteri ricercati, o `n` diventa negativo (cioè, quando si è esaminata l'intera stringa). Il lettore verifichi che il programma si comporti correttamente anche quando la stringa è vuota, o contiene solo caratteri da eliminare.

L'istruzione `continue` è legata a `break`, ma d'uso meno comune; causa l'esecuzione dell'iterazione successiva del ciclo `for`, `while` o `do`. Nel caso di `while` e `do`, ciò equivale al riesame immediato della condizione di terminazione, mentre nel caso di `for` l'esecuzione prosegue con la parte relativa all'incremento. L'istruzione `continue` non si applica all'istruzione `switch`. La presenza di `continue` all'interno di un'istruzione `switch` che sia a sua volta contenuta in un ciclo causa l'esecuzione dell'iterazione successiva del ciclo.

Come esempio ci serviamo del frammento seguente, che elabora solo gli elementi non negativi del vettore `a`, ignorando quelli negativi.

```

for (i = 0; i < n; i++) {
    if (a[i] < 0) /* ignora gli elementi negativi */
        continue;
    ... /* elabora gli elementi positivi */
}

```

L'istruzione `continue` è spesso usata quando la parte del ciclo che la segue è così complessa che l'inversione di una condizione e il rientro di un altro livello finirebbero con l'annidare troppo profondamente il programma.

### 3.8 Goto ed etichette

Il C offre il costrutto `goto`, di cui si può abusare senza limiti, e le etichette necessarie per indicare il punto del programma al quale saltare. In teoria, l'istruzione `goto` non è mai necessaria e, in pratica, è quasi sempre facile evitarla. Non abbiamo mai adoperato `goto` in questo libro.

Ciò detto, in qualche situazione `goto` non è fuori posto. Il caso più comune è la necessità di interrompere l'elaborazione all'interno di qualche costrutto annidato a livelli molto profondi nel codice, come quando si vogliono terminare prematuramente due o più cicli contemporaneamente. Non possiamo ricorrere direttamente all'istruzione `break`, perché termina solo l'iterazione più interna. Quindi:

```

for (...)
    for (... ) {
        ...
        if (disaster)
            goto error;
    }
}

```

```

    }
...

```

```

error:
raccogli i cocci

```

La struttura presentata è funzionale quando il codice per la gestione dell'errore non è banale, e quando è possibile che si verifichi lo stesso errore in diversi punti del codice.

Un'etichetta condivide la morfologia dei nomi di variabile, ed è seguita dai due punti; può essere associata a qualunque istruzione collocata nella stessa funzione che ospita goto. L'etichetta è visibile da tutta la funzione.

Volendo fare un altro esempio, si consideri il problema di determinare se due vettori a e b abbiano un elemento in comune. Una possibile codifica è:

```

for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
        if (a[i] == b[j])
            goto found;
/* nessun elemento in comune */
...
found:
/* trovato un elemento in comune: a[i] == b[j] */
...

```

Abbiamo già detto che è sempre possibile fare a meno di goto, magari al prezzo di valutare qualche condizione in più, o aggiungere delle variabili. Una versione del frammento precedente priva di goto è:

```

found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
            found = 1;
if (found)
    /* trovato un elemento in comune: a[i-1] == b[j-1] */
...
else
    /* nessun elemento in comune */
...

```

Fatte salve poche eccezioni analoghe a quelle citate, è in genere più difficile comprendere e manutenere codice che adoperi il costrutto goto. Sebbene la nostra non sia una posizione dogmatica, ci pare proprio che le istruzioni goto debbano essere usate solo occasionalmente, se non addirittura evitate.

# 4

## Funzioni e struttura dei programmi

**L**E COMPUTAZIONI LUNGHE possono essere suddivise in elementi più brevi, le funzioni, che rappresentano per l'utente la possibilità di usufruire di conoscenze già acquisite, anziché dover ricominciare ogni volta da zero. Funzioni appropriate nascondono alcuni passaggi ai brani del codice che ne possono fare a meno, migliorando così la chiarezza del programma e attenuandone la rigidità.

Il C è pensato per rendere le funzioni efficienti e facili da usare; i suoi programmi privilegiano l'uso di numerose piccole funzioni rispetto all'idea di poche funzioni ingombranti. Un programma può essere contenuto in uno o più file sorgente, che possono essere compilati singolarmente e caricati tutti in una volta, assieme alle funzioni già compilate delle librerie. Non vogliamo, tuttavia, addentrarci in questo processo, dato che i dettagli sono diversi in relazione al sistema.

La dichiarazione e la definizione delle funzioni è il settore del C a cui lo standard ANSI ha apportato i cambiamenti più evidenti. Come visto nel Capitolo 1, è ora possibile dichiarare i tipi degli argomenti quando si dichiara una funzione. Anche la sintassi della definizione di una funzione è cambiata, per far sì che dichiarazioni e definizioni siano in corrispondenza; ciò aiuta i compilatori a individuare molti più errori di quanto potessero fare prima. Inoltre, quando gli argomenti sono dichiarati in maniera corretta, le appropriate conversioni di tipo sono forzate automaticamente.

Lo standard dissipia i dubbi sulla visibilità dei nomi; nello specifico, prescrive una sola definizione per ogni oggetto esterno. L'inizializzazione è più generale: i vettori e le strutture automatici possono ora essere impostati a dei valori iniziali.

Anche il preprocessore del C ha visto aumentare le sue potenzialità. Tra esse ricordiamo: una gamma più ricca di direttive per la compilazione condizionale, un metodo per generare stringhe tra virgolette a partire dagli argomenti delle macroistruzioni, un migliore controllo del processo di espansione delle macroistruzioni.

## 4.1 Fondamenti delle funzioni

Creiamo ora un programma che visualizzi le righe del testo in ingresso che contengono un determinato “pattern” o stringa di caratteri. (Questa è una versione ridotta del programma grep di UNIX.) Per esempio, la ricerca della sequenza di lettere “ould” nel testo<sup>(1)</sup>

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

produrrà l’output

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

Il lavoro si compone di tre fasi:

```
while (c'è un'altra riga)
    if (la riga contiene una stringa della forma specificata)
        visualizzala
```

Sebbene sia certamente possibile codificare il tutto nella sola funzione main, è preferibile sfruttare la struttura naturale del problema e attribuire a ogni fase una funzione distinta. È più facile lavorare con tre segmenti piccoli che con uno grande, perché si possono relegare i dettagli irrilevanti nelle funzioni, riducendo al minimo il rischio di interazioni involontarie. Così facendo si può anche riusare il codice delle singole funzioni in altri programmi.

“while c’è un’altra riga” corrisponde a getline, una funzione che abbiamo usato nel Capitolo 1, mentre “visualizzala” è printf, di cui disponiamo grazie alla libreria. Questo vuol dire che è necessario scrivere solo il codice per decidere se la riga contiene la forma cercata.

Possiamo risolvere il problema scrivendo una funzione strindex(s,t) che restituisca la posizione o indice nella stringa s dove inizia il pattern t, o -1 se s non contiene t. Visto che nel C i vettori iniziano dalla posizione zero, gli indici saranno maggiori o uguali a zero, e quindi un valore negativo come -1 è adatto a indicare l’assenza di t in s. Se dovessimo avere bisogno di funzioni di ricerca più sofisticate, sarà sufficiente sostituire strindex; il resto può rimanere invariato. (La libreria standard fornisce una funzione strstr, simile a strindex, che però restituisce un puntatore invece di un indice.)

1. N.d.R. Celebre quartina del poeta e scienziato persiano Omar Khayyam (XI secolo D.C.), nella libera traduzione di Edward J. Fitzgerald, 1859.

Stabiliti questi punti chiave del progetto, è semplice completare il programma nei dettagli. Presentiamo il risultato finale, per dare modo al lettore di vedere come si integrano i segmenti. Il pattern da ricercare è una stringa costante, che non è certo il più generale dei meccanismi. A breve ritorneremo sull’inizializzazione dei vettori di caratteri, e nel Capitolo 5 mostreremo come rendere il pattern un parametro da impostare al momento dell’esecuzione del programma. Il lettore troverà anche una versione lievemente diversa di getline; un confronto con quella illustrata nel Capitolo 1 potrebbe essere istruttivo.

```
#include <stdio.h>
#define MAXLINE 1000 /* massima lunghezza di una riga di input */

int getline(char line[], int max)
int strindex(char source[], char searchfor[]);

char pattern[] = "ould"; /* pattern da cercare nel testo in ingresso */

/* trova tutte le righe che contengano la stringa pattern */
main()
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}

/* getline: legge una riga in ingresso, la assegna a s, ne restituisce
   la lunghezza */
int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex: restituisce l’indice di t in s, -1 se t non e’ in s */
int strindex(char s[], char t[])
{
    int i, j, k;
```

```

for (i = 0; s[i] != '\0'; i++) {
    for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
        ;
    if (k > 0 && t[k] == '\0')
        return i;
}
return -1;
}

```

Ogni definizione di funzione ha la forma

```

tipo-restituito nome-di-funzione(dichiarazioni degli argomenti)
{
    dichiarazioni e istruzioni
}

```

Diverse clausole sono facoltative; una funzione minima è

```
dummy() {}
```

che non fa e non restituisce nulla. Una funzione apparentemente inutile come questa può talora servire da segnaposto nello sviluppo di un programma. Se il tipo del valore restituito è omesso, si assume `int`.

Un programma non è altro che un insieme di definizioni di variabili e funzioni. Le funzioni comunicano tra loro tramite argomenti e valori restituiti, e attraverso variabili esterne; possono essere disposte in qualunque ordine nel file sorgente, e il programma può essere suddiviso in più file sorgente, purché nessuna funzione sia divisa in parti.

L'istruzione `return` è il meccanismo apposito per restituire un valore dalla funzione chiamata a quella chiamante. Qualunque espressione può seguire `return`:

```
return espressione;
```

dove `espressione` sarà convertita nel tipo restituito dalla funzione, se necessario. Si trova spesso `espressione` fra parentesi tonde, ma il loro uso è facoltativo.

La funzione chiamante è libera di ignorare il valore restituito. Inoltre, a `return` non deve necessariamente seguire un'`espressione`; in quel caso, nessun valore sarà restituito al chiamante. Parimenti, il controllo ritorna al chiamante senza che sia restituito alcun valore anche quando l'esecuzione della funzione chiamata raggiunge la parentesi graffa di chiusura del corpo. Una funzione che restituisca un valore da un punto del codice e nessun valore da un altro è ammessa, ma è probabilmente indice di qualche problema. Comunque, se una funzione non restituisce alcun valore, il valore della funzione dopo la sua esecuzione è senz'altro da considerarsi materiale di scarto.

Il programma per la ricerca di un pattern in una stringa restituisce un risultato da `main`, che corrisponde al numero delle occorrenze trovate. Questo valore è a disposizione dell'ambiente che ha invocato il programma.

Le dinamiche che consentono di compilare ed eseguire un programma in C distribuito in più file sorgente variano da sistema a sistema. In ambiente UNIX, per esempio, è utile il comando `cc` citato nel Capitolo 1.

Si supponga che le tre funzioni siano custodite in altrettanti file di nome `main.c`, `getline.c` e `strindex.c`. Il comando

```
cc main.c getline.c strindex.c
```

compila quindi i tre file, colloca il codice oggetto che ne risulta nei file `main.o`, `getline.o` e `strindex.o`, quindi li carica tutti in un file eseguibile chiamato `a.out`. Se si verifica un errore, poniamo in `main.c`, è possibile ricompilare solo tale file, caricando poi il risultato con i precedenti file oggetto, per mezzo del comando

```
cc main.c getline.o strindex.o
```

Il comando `cc` usa le denominazioni convenzionali “`.c`” e “`.o`” per distinguere i file sorgente dai file oggetto.

**Esercizio 4.1** Si scriva la funzione `strindex(s, t)`, che restituisca la posizione dell'occorrenza *più a destra* di `t` in `s`, o `-1` se `t` non compare in `s`.

## 4.2 Funzioni che restituiscono valori diversi da interi

Fin qui i nostri esempi di funzioni hanno restituito un intero o nulla (`void`). Ipotizziamo che una funzione debba restituire un valore di tipo diverso. Molte funzioni numeriche quali `sqrt`, `sin` e `cos` restituiscono `double`; altre funzioni specializzate restituiscono tipi diversi. Per chiarire l'argomento scriviamo la funzione `atof(s)`, che converte la stringa `s` nel suo equivalente in virgola mobile e doppia precisione. Si tratta di un'estensione di `atoi`, che abbiamo visto in diverse versioni nei Capitoli 2 e 3, in grado di gestire un segno facoltativo e il punto decimale, oltre alla presenza o assenza di una parte intera o frazionaria. La nostra versione *non* vuole essere un programma di conversione sofisticato, che richiederebbe troppo spazio. La libreria standard dispone di una funzione `atof`, dichiarata nell'intestazione `<stdlib.h>`.

In primo luogo, la stessa funzione `atof` deve dichiarare quale tipo di valore restituisce, considerato che non è un intero. Il nome del tipo precede il nome della funzione:

```
#include <cctype.h>

/* atof: converte la stringa s in un double */
double atof(char s[])
{
    double val, power;
    int i, sign;

    for (i = 0; isspace(s[i]); i++) /* ignora spazi in testa */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10 * val + (s[i] - '0');
```

```

    val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10.0;
    }
    return sign * val / power;
}

```

Cosa non meno importante, la funzione chiamante deve sapere che `atof` restituisce un valore non intero. A tal fine, si può dichiarare `atof` esplicitamente nella funzione chiamante. È quanto avviene nell'esempio seguente, una rudimentale calcolatrice (a malapena in grado di fare i conti di casa), che legge un numero (eventualmente preceduto dal segno) per ogni riga in ingresso, e li somma via via, visualizzando il risultato parziale riga per riga:

```

#include <stdio.h>

#define MAXLINE 100

/* una rudimentale calcolatrice */
main()
{
    double sum, atof(char []);
    char line[MAXLINE];
    int getline(char line[], int max);

    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%g\n", sum += atof(line));
    return 0;
}

```

La dichiarazione

```
double sum, atof(char []);
```

attesta che `sum` è una variabile di tipo `double`, e che `atof` è una funzione che si aspetta un argomento di tipo `char[]` e restituisce un valore di tipo `double`.

La funzione `atof` deve essere dichiarata e definita coerentemente. Se la medesima `atof` e la sua chiamata in `main` hanno tipi incoerenti nello stesso file sorgente, l'errore sarà rilevato dal compilatore; ma se (ipotesi più probabile) `atof` fosse compilata separatamente, l'incongruenza passerebbe inosservata: `atof` restituirebbe un valore `double`, che `main` scambierebbe per un `int`, e di conseguenza si otterrebbero risposte prive di senso.

La cosa potrebbe sembrare strana, alla luce di quanto detto sulla corrispondenza obbligatoria tra dichiarazioni e definizioni. Il motivo di questa incongruenza è che, se manca un prototipo della funzione, essa è implicitamente dichiarata dalla sua prima comparsa in un'espressione come

```
sum += atof(line)
```

Se un nome non ancora dichiarato compare in un'espressione, seguito da una parentesi aperta, il contesto lo dichiara come nome di una funzione che restituisce un intero, senza fare alcuna supposizione sui suoi argomenti. Inoltre, se una dichiarazione di funzione non contiene argomenti, come per esempio

```
double atof();
```

qualunque controllo sui parametri decade: non si fa alcuna assunzione sugli argomenti di `atof`. L'intento di questa convenzione è di favorire la compatibilità dei programmi in C vecchio stile con i nuovi compilatori, ma non è una buona idea continuare a sfruttare questa caratteristica nei nuovi programmi. Se la funzione prevede argomenti, il lettore li dichiari; in caso contrario, usi `void`.

Data `atof`, correttamente dichiarata, è possibile scrivere `atoi` (che converte una stringa in intero) come segue:

```
/* atoi: converte la stringa s in un intero sfruttando atof */
int atoi(char s[])
{
    double atof(char s[]);
    return (int) atof(s);
}
```

Si noti la struttura delle dichiarazioni e l'istruzione `return`. Il valore dell'espressione in  
`return espressione;`

è trasformato nel tipo della funzione prima che sia eseguito `return`. Pertanto, il valore di `atof`, che è `double`, è automaticamente convertito in intero quando appare in questa funzione `return`, visto che la funzione `atoi` restituisce un intero. In questo passaggio, tuttavia, si può avere perdita di informazione, motivo per cui alcuni compilatori ne danno avviso. Il costrutto `cast` sancisce esplicitamente che la conversione è desiderata dal programmatore, e sopprime ogni segnalazione di sorta.

**Esercizio 4.2** Si estenda `atof` in modo che gestisca la notazione scientifica

123.45e-6

dove un numero con virgola mobile può essere seguito da un esponente con o senza segno, il cui inizio è segnalato dalle lettere `e` o `E`.

### 4.3 Variabili esterne

Un programma in C è formato da una serie di oggetti esterni, che sono variabili oppure funzioni. L'aggettivo "esterno" è usato in contrapposizione a "interno", che descrive gli argomenti e le variabili definiti all'interno delle funzioni. Le variabili esterne sono definite al di fuori di una funzione specifica, e sono dunque potenzialmente a disposizione di molte funzioni. Le funzioni a loro volta sono sempre esterne, poiché il C non consente a una funzione di essere definita all'interno di altre funzioni. A meno di esplicite indicazioni contra-

rie, i nomi delle variabili esterne e delle funzioni denotano sempre lo stesso oggetto, indipendentemente dal punto del codice o dal file sorgente nel quale sono usati. Lo standard chiama questa proprietà “linkage esterno”. In questo senso, le variabili esterne sono analoghe ai blocchi COMMON del Fortran o alle variabili presenti nel blocco più esterno dei programmi Pascal. Più avanti si vedrà come definire variabili esterne e funzioni visibili solo all'interno di un singolo file sorgente.

Dal momento che le variabili esterne sono accessibili globalmente, rappresentano un'alternativa a parametri di funzione e valori restituiti per lo scambio di dati tra funzioni. Ogni funzione ha accesso a una variabile esterna riferendosi a essa per nome, purché tale nome sia stato dichiarato.

Qualora le variabili da condividere tra funzioni diverse siano numerose, le variabili esterne si dimostrano più opportune ed efficienti di lunghe liste di argomenti. Come ricordato nel Capitolo 1, tuttavia, questo ragionamento va applicato con cautela, poiché può influire negativamente sull'assetto dei programmi, portando a un numero eccessivo di relazioni tra funzioni.

Le variabili esterne sono utili anche in relazione alla loro maggiore visibilità e durata. Le variabili automatiche sono interne a una funzione; vengono alla luce quando la funzione entra in esecuzione, e scompaiono quando essa restituisce il controllo al chiamante. Per contro, le variabili esterne sono permanenti e conservano il valore da una chiamata della funzione alla successiva. Di conseguenza, se due funzioni devono condividere alcuni dati, ma nessuna di esse chiama l'altra, spesso la soluzione migliore è di conservare i dati in variabili esterne.

Analizziamo più in profondità questo aspetto tramite un esempio consistente. Il problema è di programmare una calcolatrice che fornisca gli operatori +, -, \* e /. La calcolatrice adotterà la notazione polacca, o prefissa, perché è più facile da trattare della usuale notazione infissa. (La notazione polacca è usata da alcune calcolatrici tascabili e da linguaggi come il Forth e il Postscript.)

Nella notazione polacca, ogni operatore fa seguito ai suoi operandi; un'espressione in notazione infissa come

$(1 - 2) * (4 + 5)$

diventa

1 2 - 4 5 + \*

Non sono necessarie parentesi; la notazione non suscita ambiguità, purché il numero di operandi di ogni operatore sia noto.

L'implementazione è semplice: ogni operando è inserito (push) in una pila (stack); quando si incontra un operatore, il numero corretto di operandi (due per gli operatori binari) viene estratto (pop) dalla pila, si applica l'operatore, e si inserisce il risultato in cima alla pila. Nell'ultimo esempio, 1 e 2 sono inseriti nella pila, e poi sostituiti dalla loro differenza, -1. Poi vengono inseriti 4 e 5, che lasciano il posto alla loro somma, 9. Quindi, il prodotto di -1 e 9, cioè -9, viene inserito al loro posto. Il valore in cima alla pila è estratto e visualizzato quando si giunge alla fine della riga in ingresso.

La struttura del programma è dunque un ciclo che effettua l'operazione richiesta dagli operatori sugli operandi che incontra:

```
while (prossimo operatore o operando non è EOF)
    if (numero)
        impila
    else if (operatore)
        estrai operandi
        esegui operazione
        impila risultato
    else if (carattere newline)
        estrai e visualizza la cima della pila
    else
        errore
```

Le operazioni per impilare ed estrarre dati da una pila sono banali; con la corretta gestione degli errori, però, diventano abbastanza lunghe da rendere preferibile dedicare una funzione separata a ogni operazione. È anche consigliabile prevedere una funzione distinta per la lettura dell'operatore o dell'operando successivo.

La scelta principale che non è stata affrontata finora è dove collocare la pila, vale a dire, quali funzioni possano accedervi direttamente. Una possibilità è di mantenerla all'interno di main, e passare la pila e la sua posizione corrente alle funzioni che la manipolano. D'altra parte, main non deve necessariamente conoscere le variabili che controllano la pila, in quanto si limita a impilare ed estrarre elementi tramite push() e pop(). Quindi abbiamo deciso di realizzare la pila tramite variabili esterne non visibili da main, a cui possono però accedere le funzioni push e pop.

Tradurre questo abbozzo in codice è abbastanza semplice. Se per il momento si attribuisce al programma un solo file sorgente, si avrà:

```
#include
...
#define
...
dichiarazioni di funzione per main
main() { ... }

variabili esterne per push e pop
void push(double f) { ... }
double pop(void) { ... }

int getop(char s[]) { ... }

funzioni chiamate da getop
```

Si vedrà in seguito come si potrebbe frazionare il programma in due o più file sorgente.

La funzione main è un ciclo, e contiene un grande switch che agisce sul tipo dell'operatore o dell'operando; è un impiego di switch più tipico rispetto a quello del Paragrafo 3.4.

```

#include <stdio.h>
#include <stdlib.h> /* per atof() */
#define MAXOP 100 /* dimensione massima di operandi e operatori */
#define NUMBER '0' /* indica che è stato trovato un numero */

int getop(char []);
void push(double);
double pop(void);

/* calcolatrice in notazione polacca */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("errore: divisione per zero\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("errore: comando ignoto %s\n", s);
            break;
        }
    }
    return 0;
}

```

Stante che + e \* sono operatori commutativi, l'ordine di combinazione degli operandi è irrilevante, ma riguardo a - e / è necessario distinguere l'operando di sinistra da quello di destra. In

```
push(pop() - pop()); /* ERRATO */
```

non è specificato l'ordine in cui valutare le due chiamate a pop. Si rende allora indispensabile, per garantire l'ordine corretto, che il primo valore sia collocato in una variabile provvisoria come abbiamo fatto in main.

```

#define MAXVAL 100 /* profondità massima della pila val */

int sp = 0; /* prossima posizione libera nella pila */
double val[MAXVAL]; /* la pila dei valori */

/* push: pone f in cima alla pila */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("errore: la pila e' piena, non si può inserire %g\n", f);
}

/* pop: estraе e restituisce il valore in cima alla pila */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("errore: la pila e' vuota\n");
        return 0.0;
    }
}

```

Una variabile è esterna se è definita al di fuori di una funzione. Dunque la pila val e il suo indice sp, che devono essere condivisi da push e pop, sono definiti all'esterno di queste funzioni, ma la funzione main stessa non fa riferimento alla pila o al suo indice, quindi l'implementazione può rimanere nascosta.

Passiamo a considerare l'implementazione di getop, funzione che legge l'operatore o l'operando successivo. Il suo compito è facile; la funzione salta gli spazi bianchi e i caratteri di tabulazione. Se il carattere successivo non è una cifra o un punto decimale, lo restituisce; altrimenti, memorizza in una stringa la sequenza di cifre (che potrebbe includere un punto decimale) e restituisce NUMBER, il segnale che è stato letto un numero.

```

#include <ctype.h>

int getch(void);
void ungetch(int);

```

```

/* getop: legge il prossimo operatore o operando numerico */
int getop(char s[])
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c; /* non si tratta di un numero */
    i = 0;
    if (isdigit(c)) /* legge la parte intera */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.') /* legge la parte frazionaria */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}

```

Che cosa sono `getch` e `ungetch`? Spesso un programma non riesce a stabilire che ha letto abbastanza input finché non ne ha letto troppo. Un esempio viene dalla lettura dei caratteri che formano un numero: finché non appaia qualcosa di diverso da una cifra, il numero non è completo; ma a quel punto il programma avrà letto un carattere di troppo, un carattere per cui non è preparato.

Il problema sarebbe risolto se fosse possibile “de-leggere” (“un-read”) il carattere indesiderato. In questo modo, a ogni lettura di un carattere in eccesso, il programma potrebbe reinviarlo all’input, e il resto del codice potrebbe comportarsi come se il carattere non fosse mai stato letto. Per fortuna, è facile simulare questa operazione scrivendo un paio di funzioni che interagiscano: `getch` fornisce il carattere in ingresso da esaminare; `ungetch` accantonava i caratteri da restituire al flusso in ingresso, cosicché `getch`, a ogni chiamata successiva, li rilegga prima di leggere nuovo input.

Ecco come le funzioni si coordinano: `ungetch` pone i caratteri accantonati in una porzione di memoria condivisa che funge da deposito, il buffer (un vettore di caratteri); `getch` attua un controllo sul buffer e, qualora accerti che è vuoto, chiama `getchar`. Deve anche esserci un indice che registri la posizione del carattere corrente nel buffer.

Il buffer e l’indice, essendo comuni a `getch` e `ungetch`, e dovendo mantenere i loro valori tra una chiamata e l’altra, devono essere esterni a entrambe le funzioni. Pertanto possiamo scrivere `getch`, `ungetch` e le loro variabili in comune, nel seguente modo:

```

#define BUFSIZE 100

char buf[BUFSIZE]; /* buffer per ungetch */
int bufp = 0; /* prossima posizione libera di buf */

```

```

int getch(void) /* legge un carattere, eventualmente accantonato prima */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c) /* accantonava un carattere letto */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: troppi caratteri\n");
    else
        buf[bufp++] = c;
}

```

La libreria standard ha una funzione `ungetc` che permette di accantonare un carattere letto; l’argomento sarà approfondito nel Capitolo 7. Si è qui utilizzato un vettore, anziché un singolo carattere, a favore di un’impostazione più generale.

**Esercizio 4.3** Data la struttura di base, estendere le funzionalità della calcolatrice è semplice. Il lettore aggiunga l’operatore di modulo (%) e le clausole per i numeri negativi.

**Esercizio 4.4** Si aggiungano comandi che visualizzino l’elemento in cima alla pila senza però estrarre, che lo duplichino, e che scambino i primi due elementi. Si preveda un comando per svuotare la pila.

**Esercizio 4.5** Si aggiunga l’accesso a funzioni della libreria quali `sin`, `exp` e `pow`. Si veda `<math.h>` nell’Appendice B, Paragrafo 4.

**Esercizio 4.6** Si aggiungano comandi per la gestione delle variabili. (È facile trattare 26 variabili denotate da singole lettere dell’alfabeto inglese.) Si aggiunga una variabile per il valore visualizzato per ultimo.

**Esercizio 4.7** Si scriva una funzione `ungets(s)` che possa accantonare un’intera stringa in ingresso già letta. È più opportuno che `ungets` interagisca con `buf` e `bufp`, o che si limiti a invocare `ungetch`?

**Esercizio 4.8** Supponendo che il carattere accantonato non possa essere più di uno, si modifichino `getch` e `ungetch` conseguentemente.

**Esercizio 4.9** Le funzioni `getch` e `ungetch` non gestiscono correttamente un segnale di `EOF` che sia stato accantonato. Si decida quale debba essere il loro comportamento in questa eventualità, quindi si passi alla realizzazione del progetto.

**Esercizio 4.10** Un approccio alternativo consiste nell’impiegare `getline` per leggere una intera riga in ingresso, il che rende superflue le funzioni `getch` e `ungetch`. Si adatti la calcolatrice a questo diverso approccio.

## 4.4 Regole di visibilità

Le funzioni e le variabili esterne che costituiscono un programma in C non devono essere necessariamente compilate nello stesso momento; è possibile che il codice sorgente del programma sia suddiviso in diversi file e che le funzioni realizzate in precedenza siano richiamate da apposite librerie.

Alcune questioni di rilievo sono:

- Come scrivere le dichiarazioni in maniera che le variabili siano opportunamente dichiarate già in fase di compilazione?
- Come organizzare le dichiarazioni affinché le relazioni fra le diverse parti siano collegate al momento di caricare il programma?
- Come evitare doppioni nelle dichiarazioni?
- Come inizializzare le variabili esterne?

Affronteremo questi temi attraverso una scomposizione del programma calcolatrice in diversi file. Si tratta di un programma talmente piccolo che non varrebbe la pena dividerlo, se non per illustrare i problemi che sorgono in programmi più grandi.

Il campo di visibilità (*scope*) di un nome corrisponde alla parte di programma in cui quel nome può essere usato. Per una variabile automatica dichiarata all'inizio di una funzione, il campo di visibilità è la funzione al cui interno è dichiarato il nome. Variabili locali dallo stesso nome in funzioni diverse non sono tra loro correlate, e lo stesso vale anche per i parametri di una funzione, che sono effettivamente variabili locali.

Il campo di visibilità di una variabile esterna o di una funzione si estende dal punto in cui questa è dichiarata fino alla fine del file compilato. Per esempio, se `main`, `sp`, `val`, `push` e `pop` sono dichiarate in un solo file in questo stesso ordine, ovvero:

```
main() { ... }

int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

double pop(void) { ... }
```

in seguito le variabili `sp` e `val` potranno essere usate in `push` e `pop` semplicemente nominandole: non occorrono ulteriori dichiarazioni. Né questi nomi, né `pop` e `push`, sono visibili da `main`.

D'altro canto, se c'è necessità di riferirsi a una variabile esterna prima che sia definita, oppure se essa è definita in un file sorgente diverso da quello in cui la si usa, una dichiarazione `extern` è obbligatoria.

È importante distinguere tra la *dichiarazione* di una variabile esterna e la sua *definizione*. Una dichiarazione annuncia le proprietà di una variabile (innanzitutto il suo tipo); una definizione provoca anche l'allocazione della quantità opportuna di memoria.

Se le righe

```
int sp;
double val[MAXVAL];
```

appaiono al di fuori di qualunque funzione, esse *definiscono* le variabili esterne `sp` e `val`, provocano l'allocazione di una quantità opportuna di memoria e inoltre svolgono la funzione di dichiarazione per il resto del file sorgente. Invece, le righe

```
extern int sp;
extern double val[];
```

*dichiarano* per il resto del file sorgente che `sp` è una variabile `int` e che `val` è un vettore `double` (la cui entità è determinata altrove), ma non creano le variabili, né riservano per esse spazio in memoria.

Tra tutti i file che compongono il programma sorgente deve esservi una sola *definizione* di una variabile esterna; altri file possono contenere dichiarazioni `extern` per accedervi. (Questo tipo di dichiarazioni si può trovare anche nel file che contiene la definizione.) La dimensione dei vettori deve essere specificata nella definizione, ma è facoltativa in una dichiarazione `extern`.

L'inizializzazione di una variabile esterna si accompagna esclusivamente alla sua definizione.

Benché si tratti di un assetto improbabile, si potrebbe usare un file per definire le funzioni `push` e `pop`, e un altro in cui inizializzare le variabili `val` e `sp`. Poi, per collegare le une alle altre, sarebbero necessarie queste definizioni e dichiarazioni:

```
nel file1:
extern int sp;
extern double val[];

void push(double f) { ... }

double pop(void) { ... }
```

```
nel file2:
int sp = 0;
double val[MAXVAL];
```

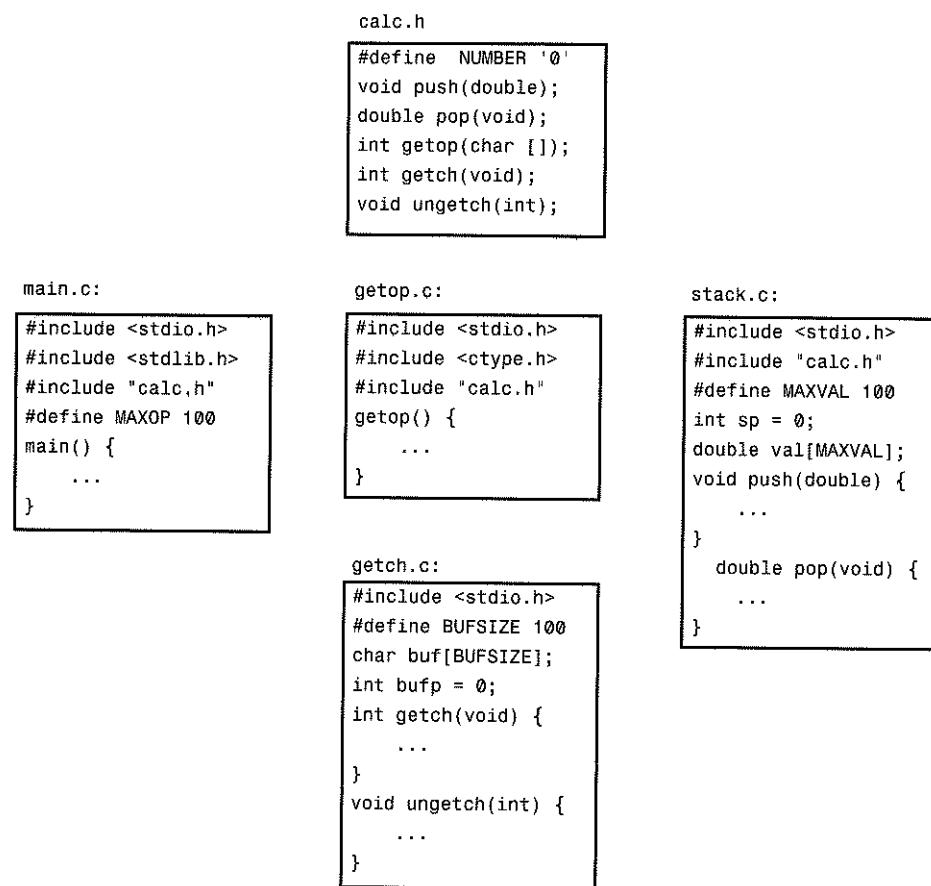
Poiché le dichiarazioni `extern` nel *file1* si trovano prima e al di fuori delle definizioni di funzione, si applicano a tutte le funzioni; una sola serie di dichiarazioni vale per l'intero *file1*. Una struttura identica occorrerebbe se le definizioni di `sp` e `val` seguissero il loro utilizzo in un file.

## 4.5 Intestazioni

Prendiamo ora in esame la suddivisione del programma calcolatrice in diversi file sorgente, un'ipotesi realistica se ognuno dei componenti fosse sensibilmente più grande. La funzione

`main` andrebbe in un file, che chiameremo `main.c`; un secondo file, `stack.c`, ospiterebbe `push`, `pop` e le relative variabili; `getop` andrebbe in un terzo, `getop.c`; infine, `getch` e `ungetch` si troverebbero in un quarto file, `getch.c`: le separiamo dalle altre perché, in un programma realistico, esse sarebbero prelevate da una libreria compilata separatamente.

L'ultimo aspetto da considerare riguarda le definizioni e le dichiarazioni condivise dai file. Se centralizziamo la loro gestione, ci potremo preoccupare di una sola copia durante lo sviluppo e la successiva manutenzione del programma. Di conseguenza riserveremo a questo materiale comune un file di intestazione (*header file*) `calc.h`, che sarà incluso ove necessario. (La direttiva `#include` è descritta nel Paragrafo 4.11.) Il programma che ne risulta ha l'aspetto mostrato nella figura seguente.



Si deve giungere a un compromesso tra la volontà di attribuire a ogni file l'accesso esclusivamente a informazioni di sua pertinenza, e l'esigenza pratica di evitare la presenza di troppe intestazioni, che renderebbe il programma più difficile da manutenere. Se il codice è di ridotte dimensioni, con ogni probabilità è preferibile una sola intestazione che contenga tutte le informazioni condivise tra le sue parti; ed è la scelta che abbiamo fatto qui. Un programma molto più grande richiederebbe maggiore organizzazione e svariate intestazioni.

## 4.6 Variabili static

Le variabili `sp` e `val` in `stack.c`, e `buf` e `bufp` in `getch.c` sono a uso esclusivo delle funzioni nei rispettivi file sorgente. La dichiarazione `static`, applicata alla dichiarazione di una variabile esterna o a una funzione, circoscrive il campo di visibilità dell'oggetto in questione al resto del file sorgente in cui risiede. La dichiarazione esterna `static` offre così un modo per nascondere nomi quali `buf` e `bufp` nella combinazione `getch-ungetch`, che devono rimanere esterni in modo da essere condivisi, pur restando invisibili agli utenti di `getch` e `ungetch`.

L'attributo `static` va premesso a una normale dichiarazione. Nel nostro esempio, se le due funzioni e le due variabili coesistono nello stesso file, come in

```

static char buf[BUFSIZE];      /* buffer per ungetch */
static int bufp = 0;           /* prossima posizione libera di buf */

int getch(void) { ... }

void ungetch(int c) { ... }

```

nessun'altra funzione avrà facoltà di accedere a `buf` e `bufp`, e i loro nomi non entreranno in conflitto con eventuali omonimi presenti in altri file del programma. Nello stesso modo possono essere nascoste le variabili impiegate da `push` e `pop` per la manipolazione della pila: è sufficiente dichiarare `sp` e `val` come `static`.

La dichiarazione esterna `static` è prevalentemente usata per le variabili, ma può anche essere applicata a funzioni. Di norma, i nomi delle funzioni sono globali, cioè visibili a qualunque brano del programma. Nel momento in cui si dichiara una funzione `static`, tuttavia, il suo nome diventa invisibile al di fuori del file in cui è dichiarata.

La dichiarazione `static` è applicabile anche alle variabili interne. Il campo di visibilità delle variabili interne `static` è limitato a una particolare funzione, proprio come per le variabili automatiche, ma a differenza di queste il loro ciclo di vita coincide con quello di tutto il programma, e non con quello della funzione che le ospita. In altre parole, le variabili interne `static` forniscono a una specifica funzione porzioni private e permanenti di memoria.

**Esercizio 4.11** Si modifichi `getop` in maniera che possa fare a meno di `ungetch`. Suggerimento: si adoperi una variabile interna `static`.

## 4.7 Variabili register

L'attributo `register` avverte il compilatore che la variabile relativa avrà un impiego massiccio. L'idea è che le variabili `register` vadano inserite nei registri della macchina, operazione che darebbe luogo a programmi più snelli e veloci: i compilatori, però, hanno facoltà di ignorare tali richieste.

Le dichiarazioni `register` si presentano così:

```

register int x;
register char c;

```

e così via. L'attributo si applica solo a variabili automatiche e ai parametri formali di una funzione. In quest'ultimo caso assume la forma

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

In pratica, vi sono restrizioni alle variabili `register` che riflettono le concrete limitazioni fisiche della macchina. Solo poche variabili per funzione possono essere contenute nei registri, e solo se il loro tipo soddisfa certi requisiti. L'abuso, o l'uso improprio, dell'attributo `register` è comunque innocuo, visto che esso viene tutt'al più ignorato. Non è possibile accedere all'indirizzo di una variabile `register` (un tema su cui si tornerà nel Capitolo 5), indipendentemente dal fatto che essa si trovi o meno in un registro. Le restrizioni in merito al numero e al tipo di variabili cui è applicabile l'attributo `register` variano a seconda della macchina.

## 4.8 Struttura a blocchi

Il C non è un linguaggio strutturato a blocchi nel senso del Pascal o linguaggi analoghi, dato che una funzione non può essere definita nell'ambito di altre funzioni. D'altra parte, le variabili possono essere definite mediante una struttura a blocchi all'interno di una funzione. Le dichiarazioni di una variabile (inizializzazione compresa) possono seguire la parentesi graffa aperta che introduce *qualunque* istruzione composta, non solo quella che dà inizio a una funzione. Le variabili così dichiarate nascondono variabili omonime eventualmente presenti in blocchi più esterni, e restano operative fino alla corrispondente parentesi chiusa. Per esempio, in

```
if (n > 0) {
    int i; /* dichiara una nuova variabile i */

    for (i = 0; i < n; i++)
        ...
}
```

il campo di visibilità della variabile `i` è dato dal ramo dell'istruzione `if` corrispondente al caso in cui `n` sia strettamente maggiore di zero; `i` non ha attinenza con nessun'altra `i` all'esterno del blocco. Una variabile automatica dichiarata e inizializzata in un blocco è inizializzata ogni volta si entra nel blocco. Una variabile `static`, invece, è inizializzata solo la prima volta che si entra nel blocco.

Le variabili automatiche, anche in veste di parametri formali, oscurano anche le variabili esterne e le funzioni omonime. Date le dichiarazioni

```
int x;
int y;

f(double x)
{
```

```
    double y;
    ...
}
```

nell'ambito della funzione `f`, le occorrenze di `x` si riferiscono al parametro, che è del tipo `double`; al di fuori di `f` esse si riferiscono all'`int` esterno. Lo stesso vale per la variabile `y`.

Esigenze di chiarezza consigliano di evitare nomi di variabile che oscurino nomi più esterni, che potrebbero causare confusione ed errori.

## 4.9 Inizializzazione

Più di una volta abbiamo citato l'inizializzazione, ma sempre a margine di altri argomenti. Riassumiamo qui alcune regole al riguardo, avendo ormai trattato le possibili categorie di variabili.

In mancanza di inizializzazione esplicita, le variabili esterne e quelle statiche sono sempre inizializzate a zero; le variabili automatiche e di classe `register` hanno valori iniziali indefiniti.

Le variabili scalari possono essere inizializzate all'atto della definizione, con un segno di uguale e un'espressione dopo il nome:

```
int x = 1;
char quote = '\'';
long day = 1000L * 60L * 60L * 24L; /* millisecondi/giorno */
```

Per le variabili esterne e statiche è d'obbligo che l'inizializzatore sia un'espressione costante; l'inizializzazione avviene una volta sola, concettualmente prima che il programma sia eseguito.

Nel caso delle variabili automatiche e di classe `register`, l'inizializzatore non è obbligatoriamente una costante: può essere qualunque espressione composta da valori definiti in precedenza, anche chiamate di funzione. Per esempio, le istruzioni di inizializzazione del programma per la ricerca binaria (Paragrafo 3.3) potrebbero essere scritte come

```
int binsearch(int x, int v[], int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

invece di

```
int low, high, mid;
low = 0;
high = n - 1;
```

A ben vedere, le inizializzazioni di variabili automatiche sono semplicemente un'alternativa breve alle istruzioni di assegnamento, e la scelta tra le due possibilità è sostanzialmente una questione di gusto. Noi abbiamo solitamente preferito gli assegnamenti esplicativi, perché è più difficile individuare gli inizializzatori nelle dichiarazioni; la loro collocazione, inoltre, è

spesso troppo distante dal punto del codice in cui quei valori sono effettivamente usati.

Un vettore può essere inizializzato facendo seguire alla sua dichiarazione una lista di inizializzatori racchiusi tra parentesi graffe e separati da virgole, per esempio, per inizializzare il vettore days che elenca il numero di giorni di ogni mese:

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31 };
```

Quando è omessa la dimensione del vettore, il compilatore risalirà alla sua lunghezza contando gli inizializzatori, che in questo caso sono 12.

Se un vettore non ha un numero sufficiente di inizializzatori, gli elementi mancanti si assumono essere zero per le variabili esterne, automatiche e statiche. È un errore elencare troppi inizializzatori. La ripetizione di un inizializzatore non è fattibile, come non lo è inizializzare per primo un elemento intermedio del vettore saltando i precedenti.

I vettori di caratteri sono un caso particolare; come inizializzatore, al posto della notazione con graffe e virgole, è possibile usare una stringa:

```
char pattern = "ould";
```

è un'abbreviazione dell'equivalente

```
char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

In questo esempio, la dimensione del vettore è cinque (quattro caratteri più lo '\0' finale).

## 4.10 Ricorsione

Le funzioni del C contemplano l'utilizzo ricorsivo: una funzione può chiamare se stessa, in maniera diretta o indiretta. Si consideri per esempio la visualizzazione di un numero come stringa di caratteri. Come già accennato, le cifre vengono prodotte nell'ordine errato: prima le unità, poi le decine, le centinaia, e così via; occorre quindi visualizzarle nell'ordine inverso.

Le soluzioni a questo problema sono due. La prima è di memorizzare le cifre in un vettore, non appena siano disponibili, per poi visualizzarle nell'ordine inverso, come abbiamo fatto nella funzione `itoa` nel Paragrafo 3.6. La seconda è la soluzione ricorsiva, in cui `printd` chiama dapprima se stessa per trattare le cifre d'ordine superiore (poniamo, le centinaia e le decine), quindi visualizza le unità. Tuttavia, anche questa versione può fallire se il numero in ingresso è il massimo valore negativo ammesso.

```
#include <stdio.h>

/* printd: visualizza l'intero n come sequenza di caratteri */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

Quando una funzione chiama se stessa ricorsivamente, a ogni invocazione si genera una nuova copia di tutte le variabili automatiche, a prescindere dal loro uso precedente. Pertanto, in `printd(123)` la prima `printd` riceve l'argomento `n = 123`; passa 12 a un secondo esemplare di `printd`, che a sua volta passa 1 a un terzo esemplare, che visualizza 1, e restituisce il controllo al secondo esemplare; esso visualizza 2, e ritorna il controllo al primo esemplare di `printd`, che visualizza 3 e termina.

Un altro buon esempio di ricorsione è quicksort, un algoritmo di ordinamento concepito da C. A. R. Hoare nel 1962. Dato un vettore, si individua un elemento che funga da spartiacque. Gli elementi sono poi suddivisi in due sottoinsiemi: quelli minori dello spartiacque, e quelli uguali o maggiori. Lo stesso procedimento si applica ricorsivamente ai due sottoinsiemi; quando uno di essi ha meno di due elementi non necessita di alcun ordinamento, e ciò interrompe la ricorsione.

La nostra versione di quicksort non è la più veloce possibile, ma è tra le più semplici. Useremo come spartiacque l'elemento centrale del vettore corrente.

```
/* qsort: ordina v[left]...v[right] in ordine crescente */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right) /* non fa nulla se il vettore contiene */
        return;           /* meno di due elementi */
    swap(v, left, (left + right)/2); /* sposta lo spartiacque */
    last = left;          /* in v[0] */
    for (i = left + 1; i <= right; i++) /* suddivide gli elementi */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last); /* riporta lo spartiacque al suo posto */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Abbiamo spostato l'operazione di scambio di due elementi in una funzione separata `swap`, dal momento che compare tre volte in `qsort`.

```
/* swap: scambia di posto v[i] and v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

La libreria standard possiede una versione di `qsort` in grado di ordinare oggetti di qualunque tipo.

In generale, la ricorsione non garantisce prestazioni superiori rispetto a una versione iterativa. Essa non porta neppure ad alcun risparmio di memoria, visto che il sistema la implementa tramite una pila contenente i valori da elaborare. Il codice ricorsivo è comunque più compatto, e spesso è notevolmente più facile da scrivere e da capire del suo equivalente iterativo. La ricorsione si rivela particolarmente adatta nel caso di strutture di dati definite ricorsivamente, come gli alberi; è quanto vedremo nel Paragrafo 6.5.

**Esercizio 4.12** Si adatti `printd` così da scrivere una versione ricorsiva di `itoa`; si trasformi, cioè, un intero in una stringa che lo rappresenti sfruttando la ricorsione.

**Esercizio 4.13** Si scriva una versione ricorsiva della funzione `reverse(s)`, che inverte la stringa `s` senza eseguirne copia.

## 4.11 Il preprocessore del C

Il C offre determinate funzionalità grazie a un preprocessore che, sul piano concettuale, costituisce una prima fase separata della compilazione. Le caratteristiche più usate sono due: `#include`, che permette di incorporare il contenuto di un file specificato durante la compilazione, e `#define`, che permette di sostituire un dato simbolo con una sequenza arbitraria di caratteri. Altre caratteristiche trattate in questo paragrafo sono la compilazione condizionale e le macroistruzioni con argomenti.

### 4.11.1 Inclusione di file

L'inclusione di file agevola, tra le altre cose, l'organizzazione delle istruzioni `#define` e delle dichiarazioni. Ogni riga di testo nella forma

```
#include "nomedelfile"
o
#include <nomedelfile>
```

è sostituita dai contenuti del file `nomedelfile`. Se `nomedelfile` si trova tra virgolette, la ricerca del file inizia tipicamente dal punto in cui risiede il file sorgente; se non si trova lì, o se il nome è racchiuso tra < e >, la ricerca si attiene a una regola definita dall'implementazione per reperire il file. Un file incluso può a sua volta contenere direttive `#include`.

Spesso si trovano numerose direttive `#include` all'inizio di un file sorgente, per allegare istruzioni `#define` e dichiarazioni `extern` che sono in comune con altre parti del programma, o per accedere a dichiarazioni di prototipo di funzione da intestazioni come `<stdio.h>`. (A rigor di termini, non deve trattarsi imperativeamente di file; i dettagli su come accedere alle intestazioni dipendono dall'implementazione.)

La direttiva `#include` è lo strumento preferenziale per collegare le dichiarazioni in un programma di grandi dimensioni. Assicura che a tutti i file sorgente siano fornite le medesime definizioni e dichiarazioni, eliminando così un genere di errore particolarmente insidioso. Va da sé che quando un file incluso viene modificato, tutti i file che da esso dipendono devono essere ricompilati.

### 4.11.2 Sostituzione di macroistruzioni

Una definizione nella forma:

```
#define nome testo sostitutivo
```

richiede una sostituzione della natura più semplice; a ogni occorrenza dell'oggetto `nome` deve subentrare `testo sostitutivo`. Si tratta del tipo più elementare di macroistruzione, abbreviato per semplicità in macro. Il nome in una direttiva `#define` ha la stessa morfologia del nome di una variabile; il testo sostitutivo è arbitrario. Esso è costituito, in genere, dal testo residuo della riga, ma una definizione lunga può continuare su diverse righe, fatto che si segnala posizionando \ alla fine di ogni riga da proseguire. Il campo di visibilità del nome definito con `#define` va dal suo punto di definizione al termine del file sorgente. Una definizione può usare definizioni precedenti. Le sostituzioni coinvolgono solo token (si veda l'Appendice A, paragrafo A.2.1) e non riguardano le stringhe tra virgolette. Per esempio, se `BORSE` è un nome definito, la sostituzione non avrebbe luogo nei confronti di `printf ("BORSE")` o di `PORTABORSE`.

Qualunque nome è associabile a qualsivoglia testo sostitutivo. L'esempio

```
#define forever for(;;) /* ciclo infinito */
```

definisce un termine nuovo, `forever`, che denota un ciclo infinito.

È anche possibile definire macro con argomenti, per cui il testo sostitutivo può cambiare in rapporto a chiamate diverse della macro. A titolo di esempio, definiamo una macro di nome `max`:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Nonostante sembri in apparenza una chiamata di funzione, ogni esemplare di `max` è sostituito dal codice opportuno a cura del preprocessore; nessuna chiamata di funzione ha luogo. Ogni occorrenza di un parametro formale (qui `A` o `B`) lascerà il posto all'argomento attuale corrispondente. Pertanto la riga

```
x = max(p+q, r+s);
```

sarà sostituita dalla riga

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Purché gli argomenti siano trattati coerentemente, questa macro sarà adatta a tutti i tipi di dati; non è richiesta una tipologia specifica per diversi tipi di dati, come nel caso delle funzioni.

Se si osserva il risultato delle sostituzioni eseguite su `max`, si noteranno alcune insidie. Le espressioni sono valutate due volte, il che causa problemi in presenza di effetti collaterali come quelli discussi nel Paragrafo 2.12. Per esempio,

```
max(i++, j++) /* ERRATO */
```

aumenterà di due unità il valore più grande. Bisogna prestare attenzione anche alle parentesi, per assicurarsi che l'ordine di valutazione sia rispettato; basti pensare a cosa accade quando la macro

```
#define square(x) x * x /* ERRATO */
```

è invocata come `square(z+1)`.

Tuttavia, le macro sono strumenti validi. Un esempio pratico viene da `<stdio.h>`, in cui le funzioni `getchar` e `putchar` sono spesso definite come macro per evitare il rallentamento indotto da una chiamata di funzione per ogni carattere da elaborare. Solitamente anche le funzioni in `<ctype.h>` sono implementate come macro.

Si può annullare l'effetto di un'eventuale definizione precedente tramite la direttiva `#undef`, solitamente al fine di garantire che un certo nome denoti una funzione, e non una macro:

```
#undef getchar

int getchar(void) { ... }
```

I parametri formali all'interno di stringhe tra virgolette non subiscono sostituzione. Se, invece, un nome di parametro è preceduto da un `#` nel testo sostitutivo, la combinazione sarà trasformata in una stringa tra virgolette che contiene l'argomento attuale. Questa caratteristica può essere combinata con la concatenazione delle stringhe per realizzare, per esempio, una macro utile nell'esaminare la correttezza dei programmi (cioé, per il debugging del codice):

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

Invocando

```
dprint(x/y);
```

il preprocessore esegue la sostituzione

```
printf("x/y" " = %g\n", x/y);
```

e le stringhe si concatenano, con effetto equivalente a

```
printf("x/y = %g\n", x/y);
```

Nell'argomento attuale, ogni `"` è sostituito da `\`, e ogni `\` da `\\`; ne consegue una costante stringa corretta.

L'operatore del preprocessore `##` fornisce un modo per concatenare gli argomenti attuali durante la sostituzione della macro. Un parametro che nel testo sostitutivo sia in posizione adiacente a `##` viene sostituito dall'argomento attuale, `##` è rimosso insieme allo spazio bianco circostante, e il risultato è riesaminato. Per esempio, la macro `paste` concatena i suoi due argomenti:

```
#define paste(front, back) front ## back
```

di modo che `paste(nome, 1)` equivale a `nome1`.

Le regole per il trattamento di occorrenze di `##` annidate sono intricate; approfondimenti si possono trovare nell'Appendice A.

**Esercizio 4.14** Si definisca una macro `swap(t,x,y)` che scambi due argomenti di tipo `t`. (Sarà di aiuto una struttura a blocchi.)

### 4.11.3 Inclusione condizionale

È possibile controllare l'attività del preprocessore mediante istruzioni condizionali valutate durante la fase di pre-elaborazione. Così facendo si include codice in modo selettivo, a seconda del valore delle condizioni esaminate nella fase di compilazione.

La direttiva `#if` valuta un'espressione costante intera (che non può però comprendere istruzioni `sizeof`, `cast` o costanti `enum`). Se l'espressione è diversa da zero, le righe a seguire sono accolte al codice da compilare, finché non appaia una direttiva `#endif`, `#elif` o `#else`. (La direttiva per il preprocessore `#elif` è analoga alla clausola `else if`.) L'espressione `defined(nome)` in una direttiva `#if` vale 1 se `nome` è stato definito, e 0 altrimenti.

Se, per esempio, si vuole garantire che i contenuti del file `hdr.h` siano inclusi una volta sola, tali contenuti vanno inseriti all'interno di un'espressione condizionale, quale:

```
#if !defined(HDR)
#define HDR

/* qui i contenuti di hdr.h */

#endif
```

La prima inclusione di `hdr.h` definisce il nome `HDR`; le inclusioni successive salteranno a `#endif`, perché il nome è già definito. Si può applicare una tecnica simile per evitare di includere i file più volte. Grazie a essa, inoltre, ogni intestazione può a sua volta includere le altre intestazioni da cui dipende, sollevando l'utente dalla gestione delle interdipendenze.

Questa sequenza esamina il nome `SYSTEM` per decidere quale versione di una data intestazione debba essere inclusa:

```
#if SYSTEM == SYSV
#define HDR "sysv.h"
#elif SYSTEM == BSD
#define HDR "bsd.h"
#elif SYSTEM == MSDOS
#define HDR "msdos.h"
#else
#define HDR "default.h"
#endif
#include HDR
```

Le direttive `#ifdef` e `#ifndef` sono costrutti specializzati che verificano se un nome è definito. Il primo esempio di `#if` è equivalente a

```
#ifndef HDR
#define HDR

/* qui i contenuti di hdr.h */

#endif
```

## Puntatori e vettori

**U**N PUNTATORE È UNA VARIABILE che contiene l'indirizzo di un'altra variabile. I puntatori sono largamente utilizzati nel C, in parte perché talvolta sono l'unico strumento per realizzare una computazione, in parte perché i benefici di compattezza ed efficienza che conferiscono ai programmi sono maggiori rispetto ad altri metodi. Puntatori e vettori sono strettamente connessi; il capitolo illustra anche questa relazione e spiega come metterla a frutto.

I puntatori sono stati paragonati all'istruzione `goto` in quanto mezzo portentoso per dare vita a programmi indecifrabili. Senza dubbio, questo è vero quando se ne fa un uso distratto, e lo conferma la facilità con cui si ottengono puntatori che denotano aree di memoria inattese. Con la dovuta attenzione, tuttavia, è possibile conseguire risultati di chiarezza e semplicità. Su questo aspetto ci soffermeremo.

L'innovazione fondamentale dell'ANSI C è di rendere esplicite le regole sui puntatori, di fatto decretando l'obbligatorietà di ciò che i buoni programmati già praticano e i buoni compilatori già richiedono. Inoltre, il tipo `void *` (puntatore a `void`) sostituisce `char *` come tipo prescelto per un puntatore generico.

### 5.1 Puntatori e indirizzi

Partiamo da una semplice illustrazione che rappresenta come è organizzata la memoria. Una macchina tipica è dotata di una collezione di celle o locazioni di memoria numerate

consecutivamente da indirizzi, soggetto alla manipolazione singola o in gruppi contigui. Una situazione comune è che ogni byte può rappresentare un tipo `char`, una coppia di celle da un byte può essere trattata come un intero `short` e quattro byte adiacenti come un intero `long`. Un puntatore è un gruppo di celle (spesso due o quattro) contenente un indirizzo. Dunque se `c` è `char` e `p` è un puntatore che rimanda a esso, potremmo raffigurare la situazione in questo modo:



L'operatore unario `&` fornisce l'indirizzo di un oggetto, pertanto l'istruzione

```
p = &c;
```

assegna l'indirizzo di `c` alla variabile `p`; si dice che `p` "puanta a" `c`. L'operatore `&` si applica solo a oggetti riposti in memoria: variabili ed elementi di vettori. Non può applicarsi a espressioni, costanti o variabili di classe `register`.

L'operatore unario `*` è detto di *indirezione* o *deriferimento* ("indirection" o "dereferencing"); se applicato a un puntatore, accede all'oggetto indicato dal puntatore. Si supponga che `x` e `y` siano interi e che `ip` sia un puntatore a un intero. Questo brano artificioso mostra come dichiarare un puntatore e come usare `&` e `*`:

```
int x = 1, y = 2, z[10];
int *ip; /* ip è un puntatore a un intero */

ip = &x; /* ip adesso punta a x */
y = *ip; /* y vale adesso 1 */
*ip = 0; /* x vale adesso 0 */
ip = &z[0]; /* ip adesso punta a z[0] */
```

Delle variabili `x`, `y` e `z` si è già trattato. La dichiarazione del puntatore `ip`,

```
int *ip;
```

afferma che l'espressione `*ip` è di tipo `int`, e dunque va intesa come promemoria sulla natura di `ip`. La sintassi della dichiarazione di una variabile è simile a quella di espressioni in cui la variabile potrebbe apparire. Tale logica si estende anche alle dichiarazioni di funzione:

```
double *dp, atof(char *);
```

dice che, in un'espressione, `*dp` e `atof(s)` assumono valori del tipo `double`, e che l'argomento di `atof` è un puntatore a `char`.

Non sfugga l'implicazione per cui un puntatore è obbligato a indicare oggetti di un determinato genere: ogni puntatore punta a un tipo specifico di dati. (C'è un'eccezione: un "puntatore a void" si usa per contenere qualunque tipo di puntatore ma non consente, di per sé, il deriferimento. Si tornerà sull'argomento nel Paragrafo 5.11.)

Se `ip` punta all'intero `x`, allora `*ip` può comparire dovunque lo possa fare `x`, cosicché

```
*ip = *ip + 10;
```

aumenta `*ip` di 10 unità.

Gli operatori unari `*` e `&` hanno diritto di precedenza sugli operatori aritmetici, pertanto l'assegnamento

```
y = *ip + 1
```

ha l'effetto di prendere l'oggetto puntato da `ip`, incrementarlo di 1 e assegnare il risultato a `y`, mentre

```
*ip += 1
```

incrementa l'oggetto puntato da `ip`, così come

```
++*ip
```

e

```
(*ip)++
```

Le parentesi sono necessarie in questo ultimo esempio, altrimenti l'espressione incrementerebbe `ip` invece dell'oggetto da esso puntato, visto che gli operatori unari come `*` e `++` associano da destra a sinistra.

Infine i puntatori, in quanto variabili, possono essere utilizzati senza ricorrere al deriferimento. Per esempio, se `iq` è un altro puntatore a `int`,

```
iq = ip
```

copia i contenuti di `ip` in `iq`, con la conseguenza che `iq` punterà a ciò che è puntato da `ip`.

## 5.2 Puntatori e argomenti delle funzioni

Dal momento che nel C gli argomenti sono passati alle funzioni per valore, per la funzione chiamata non esiste un modo diretto di alterare una variabile della funzione chiamante. Un algoritmo di ordinamento, per esempio, potrebbe dover scambiare due elementi fuori posto tramite una funzione di nome `swap`. Non è sufficiente scrivere

```
swap(a, b);
```

dove la funzione `swap` è definita come

```
void swap(int x, int y) /* ERRATO */
{
    int temp;

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

A causa del passaggio dei parametri per valore, swap non è in grado di agire sugli argomenti *a* e *b* nella funzione da cui è stata chiamata. Il codice appena riportato scambia semplicemente delle *copie* di *a* e *b*.

Per ottenere l'effetto desiderato, il chiamante deve passare i *puntatori* ai valori da cambiare:

```
swap(&a, &b);
```

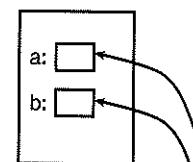
Dato che l'operatore `&` genera l'indirizzo di una variabile, `&a` è un puntatore ad *a*. La funzione swap deve dichiarare che i parametri sono puntatori: l'accesso agli operandi avverrà indirettamente tramite essi.

```
void swap(int *px, int *py) /* scambia *px e *py */
{
    int temp;

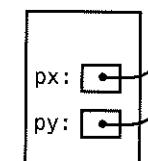
    temp = *px;
    *px = *py;
    *py = temp;
}
```

Graficamente:

nel chiamante:



in swap:



Argomenti di tipo puntatore come questi mettono una funzione in condizione di raggiungere e modificare gli oggetti appartenenti alla funzione chiamante. Si consideri, a titolo di esempio, una funzione getint che converta un flusso arbitrario di caratteri in ingresso in valori interi, un intero per chiamata. Essa deve restituire un intero, nonché indicare la terminazione del file (`EOF`) quando opportuno, ma le due cose non possono essere entrambe eseguite tramite l'ordinario meccanismo di restituzione di un valore da parte di una funzione; infatti, qualunque valore abbia `EOF`, esso potrà coincidere con il valore di un intero in ingresso.

Una soluzione consiste nel progettare getint in maniera che restituisca come valore l'avviso di terminazione del file, usando invece un argomento puntatore per memorizzare l'intero convertito nella funzione chiamante. Questo è il metodo adottato anche da scanf; si veda il Paragrafo 7.4.

Il ciclo che presentiamo di seguito imposta un vettore di interi, chiamando getint:

```
int n, array[SIZE], getint(int *);

for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
;
```

Ogni chiamata imposta il valore di `array[n]` al successivo intero in ingresso, e incrementa *n*. Si osservi che è essenziale passare l'indirizzo di `array[n]` a getint, che altrimenti non potrebbe rendere noto al chiamante l'intero convertito.

La nostra versione di getint restituisce `EOF` per segnalare la fine del file, zero se il successivo dato in ingresso non è un numero, e un valore positivo nel caso di un numero valido.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getint: legge il prossimo intero in ingresso, e lo memorizza in *pn */
int getint(int *pn)
{
    int c, sign;

    while (isspace(c = getch())) /* ignora gli spazi */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-')
        ungetch(c); /* non si tratta di un numero */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c), c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

All'interno di getint, `*pn` è impiegata come una normale variabile intera. Abbiamo usato anche le funzioni `getch` e `ungetch`, descritte nel Paragrafo 4.3, così da poter riconsegnare al flusso in ingresso il carattere in più che deve essere letto.

**Esercizio 5.1** Per come è scritta, `getint` tratta un + o un - non seguiti da una cifra come una rappresentazione valida di zero. Si apportino le modifiche necessarie per restituire al flusso in ingresso il segno + o -, qualora esso non sia seguito da una cifra.

**Esercizio 5.2** Si scriva `getfloat`, l'analogo con virgola mobile di `getint`. Quale tipo restituisce `getfloat`?

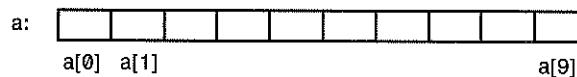
### 5.3 Puntatori e vettori

Nel C sussiste una relazione tra puntatori e vettori talmente forte da richiedere che i due argomenti vengano presentati contemporaneamente. Ogni operazione ottenibile tramite l'accesso a un vettore per mezzo di un indice può essere realizzata anche con i puntatori. L'impiego dei puntatori garantisce generalmente maggiore velocità d'esecuzione, ma risulta per certi versi più difficile da capire, almeno ai neofiti.

La dichiarazione

```
int a[10];
```

definisce un vettore di grandezza 10, cioè un blocco di 10 oggetti consecutivi denominati `a[0]`, `a[1]`, ..., `a[9]`.



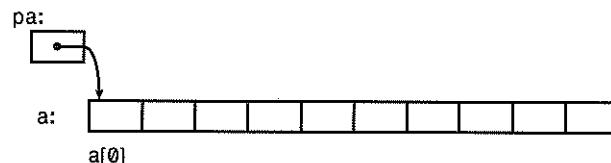
La notazione `a[i]` denota l' $i$ -esimo elemento del vettore. Se `pa` è un puntatore a un intero, dichiarato come

```
int *pa;
```

ne consegue che l'assegnamento

```
pa = &a[0];
```

impone a `pa` di puntare all'elemento in posizione zero di `a`; in altri termini, `pa` contiene l'indirizzo di `a[0]`.



Ora l'assegnamento

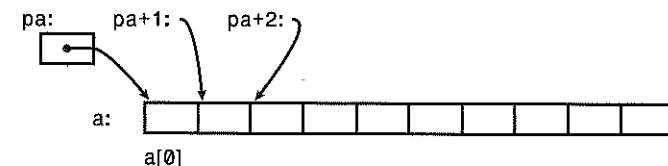
```
x = *pa;
```

copierà i contenuti di `a[0]` in `x`.

Se `pa` punta a un elemento di un vettore, per definizione `pa+1` punta all'elemento successivo, `pa+i` punta all' $i$ -esimo elemento dopo `pa`, e `pa-i` punta all' $i$ -esimo elemento prima di `pa`. Quindi, se `pa` punta ad `a[0]`,

`* (pa+1)`

si riferisce ai contenuti di `a[1]`, `pa+i` è l'indirizzo di `a[i]`, e `*(pa+i)` dà il contenuto di `a[i]`.



Queste osservazioni valgono indipendentemente dal tipo o dalle dimensioni delle variabili nel vettore `a`. Il significato di "aggiungere 1 a un puntatore", e in generale di tutta l'aritmetica dei puntatori, è che `pa+i` punta all'oggetto successivo, e `pa+i` all' $i$ -esimo oggetto dopo `pa`.

La corrispondenza tra l'aritmetica degli indici e quella dei puntatori è molto stretta. Per definizione, il valore di una variabile o di un'espressione di tipo vettore è l'indirizzo dell'elemento in posizione zero del vettore. Per cui, dopo l'assegnamento

```
pa = &a[0];
```

`pa` e `a` assumono valori identici. Dato che il nome di un vettore designa la posizione dell'elemento iniziale (quello cioè il cui indice è zero), l'assegnamento `pa=&a[0]` si può riscrivere come

```
pa = a;
```

Alquanto più sorprendente, almeno a prima vista, è il fatto che un riferimento ad `a[i]` si può anche scrivere come `*(a+i)`. Nel valutare `a[i]`, il C lo trasforma immediatamente in `*(a+i)`; le due forme si equivalgono. Se si applica l'operatore `&` a entrambi i membri di questa equivalenza, si evince che pure `&a[i]` e `a+i` sono sinonimi: `a+i` è l'indirizzo dell' $i$ -esimo elemento dopo `a`. Come rovescio della medaglia, si può correttamente applicare un indice a un puntatore `pa`, in quanto `pa[i]` è identico a `*(pa+i)`. In breve, un'espressione composta da un vettore e un indice equivale a una in forma di puntatore e scostamento. La variabile `i` che compare in un'espressione come `pa+i` è detta "scostamento" (dell'indirizzo `pa`), dall'inglese *offset*; l'indirizzo `pa` è detto invece "base".

Tra il nome di un vettore e un puntatore corre una differenza che bisogna tenere a mente. Un puntatore è una variabile, quindi `pa=a` e `pa++` sono espressioni corrette. Ma il nome di un vettore non è una variabile; costrutti quali `a=pa` e `a++` sono sintatticamente scorretti.

Quando il nome di un vettore è passato a una funzione, ciò che si comunica è la posizione del suo primo elemento. Nell'ambito della funzione chiamata, questo argomento è una variabile locale, sicché un parametro che sia il nome di un vettore è un puntatore, il che significa una variabile che contiene un indirizzo. Possiamo avvalerci di questa proprietà per scrivere un'altra versione di `strlen`, che calcola la lunghezza di una stringa.

```
/* strlen: restituisce la lunghezza della stringa s */
int strlen(char *s)
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

Poiché *s* è un puntatore, incrementarlo è perfettamente legale; *s**++* non influisce sulla stringa di caratteri della funzione che ha chiamato *strlen*, ma incrementa solo la copia privata del puntatore posseduta da *strlen*. Ciò vuol dire che chiamate come

```
strlen("ciao, mondo"); /* costante di tipo stringa */
strlen(array);          /* char array[100]; */
strlen(ptr);            /* char *ptr; */
```

funzionano tutte.

In qualità di parametri formali nella definizione di una funzione,

```
char s[];
```

e

```
char *s;
```

sono equivalenti; la nostra preferenza va al secondo perché afferma più chiaramente che il parametro è un puntatore. Quando un nome di vettore è passato a una funzione, questa può ritenere a suo piacimento di avere a che fare o con un vettore o con un puntatore, e trattarlo di conseguenza. Può persino utilizzare entrambe le notazioni qualora siano chiare e opportune.

È possibile passare solo parte di un vettore a una funzione, passando un puntatore all'inizio di un sottovettore. Per esempio, se *a* è un vettore, tanto

```
f(&a[2])
```

quanto

```
f(a+2)
```

passano alla funzione *f* l'indirizzo del sottovettore che comincia da *a*[2]. All'interno di *f*, la dichiarazione del parametro può essere

```
f(int arr[]) { ... }
```

oppure

```
f(int *arr) { ... }
```

Dunque il fatto che il parametro rimandi a una parte di un vettore più esteso non ha conseguenze per quanto attiene a *f*.

Se si è certi dell'esistenza di elementi del vettore in quelle posizioni, è anche possibile usare indici negativi; *p*[-1], *p*[-2], e così via sono espressioni corrette sotto il profilo sintattico, e si riferiscono agli elementi che precedono *p*[0]. Naturalmente, non è ammesso riferirsi a oggetti che non si trovano all'interno del vettore.

## 5.4 Aritmetica degli indirizzi

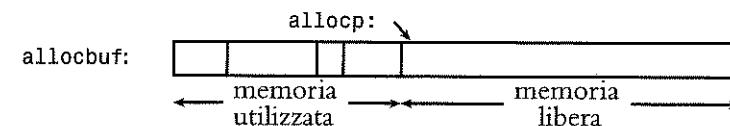
Se *p* è un puntatore a un elemento di un vettore, *p**++* incrementa *p* in modo che denoti l'elemento successivo, e *p**+i* lo incrementa affinché indichi i elementi dopo quello originalmente puntato. Questi costrutti e altri simili sono le forme più semplici dell'aritmetica dei puntatori o degli indirizzi.

Il C ha un'impostazione coerente e solida nei confronti dell'aritmetica degli indirizzi; uno dei suoi punti di forza è proprio l'integrazione di puntatori, vettori e aritmetica degli indirizzi. Ne diamo un esempio presentando un rudimentale allocatore di memoria. Esso comprende due funzioni: la prima, *alloc(n)*, restituisce un puntatore *p* a *n* locazioni consecutive, ognuna delle dimensioni di un carattere, che il chiamante di *alloc* può usare per la memorizzazione; la seconda, *afree(p)*, libera la memoria acquisita in questo modo per riutilizzarla in seguito. Si tratta di procedure "rudimentali" perché le chiamate ad *afree* devono essere fatte nell'ordine inverso delle chiamate ad *alloc*. Ciò significa che la memoria gestita da *alloc* e *afree* è strutturata come una pila, ossia una lista gestita secondo la politica LIFO (Last-In, First-Out: l'ultimo arrivato è anche il primo a essere servito). La libreria standard offre funzioni analoghe chiamate *malloc* e *free*, non soggette a simili restrizioni; nel Paragrafo 8.7 vedremo come implementarle.

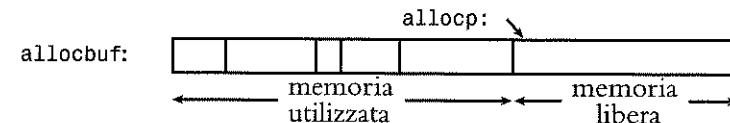
L'implementazione più facile prevede che *alloc* consegni segmenti di un vettore di caratteri di grandi dimensioni, che chiameremo *allocbuf*. L'accesso al vettore è riservato ad *alloc* e *afree*. Poiché queste ultime sfruttano puntatori, e non indici di vettori, non è necessario che altre funzioni conoscano il nome del vettore, che può essere dichiarato di classe *static* nel file sorgente che contiene *alloc* e *afree*, con l'effetto di risultare invisibile al resto del codice. Nelle implementazioni reali, un tale vettore può persino restare anonimo; lo si potrà ottenere, per esempio, invocando *malloc*, oppure dietro richiesta al sistema operativo di un puntatore a un qualche blocco di memoria anonima.

L'altra informazione necessaria è quale porzione di *allocbuf* sia già in uso. Ci serviamo di un puntatore, chiamato *allocp*, che indica il primo elemento libero. Quando ad *alloc* sono richiesti *n* caratteri, essa verifica che lo spazio residuo in *allocbuf* sia sufficiente; se è così, restituisce il valore corrente di *allocp* (cioè, l'inizio del blocco libero), quindi lo incrementa di *n* per farlo puntare alla successiva area libera; se non c'è spazio disponibile, restituisce zero. La funzione *afree(p)* si limita a impostare *allocp* a *p*, sempre che quest'ultimo valore cada all'interno di *allocbuf*.

prima di invocare *alloc*:



dopo aver invocato *alloc*:



```
#define ALLOCSIZE 10000 /* dimensione dello spazio disponibile */

static char allocbuf[ALLOCSIZE]; /* memoria gestita da alloc */
static char *allocp = allocbuf; /* posizione libera corrente */

char *alloc(int n) /* restituisce un puntatore a n caratteri */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* memoria disponibile */
        allocp += n;
        return allocp - n; /* posizione precedente */
    } else /* non c'e' abbastanza spazio */
        return 0;
}

void afree(char *p) /* libera la zona di memoria puntata da p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}

```

Un puntatore può essere inizializzato esattamente come le altre variabili, malgrado di norma gli unici valori significativi siano zero o un'espressione con indirizzi di dati definiti in precedenza e di tipo appropriato. La dichiarazione

```
static char *allocp = allocbuf;
```

definisce `allocp` come puntatore a un carattere e lo inizializza affinché punti all'inizio di `allocbuf`, che è la prima posizione disponibile all'avvio del programma.

Avremmo potuto anche scrivere

```
static char *allocp = &allocbuf[0];
```

dato che il nome del vettore è l'indirizzo dell'elemento in posizione zero.

La condizione

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* memoria disponibile */
```

controlla lo spazio disponibile per capire se la richiesta di `n` caratteri può essere soddisfatta. In caso affermativo, il nuovo valore di `allocp` dopo l'allocazione supererà al più di uno la fine di `allocbuf`. Se la richiesta può essere esaudita, `alloc` restituisce un puntatore all'inizio di un blocco di caratteri (si noti la dichiarazione della funzione stessa). In caso contrario, `alloc` deve segnalare in qualche modo la mancanza di spazio. Il C garantisce che zero non sia mai un indirizzo valido per i dati, pertanto il valore zero è adatto a segnalare un evento anomalo come, qui, la mancanza di spazio.

Puntatori e interi non sono intercambiabili. Zero è l'unica eccezione: la costante zero può essere assegnata a un puntatore, e un puntatore può essere confrontato con la costante zero. In luogo di zero è spesso usata la costante `NULL`, come promemoria per chiarire che si tratta di un valore speciale per un puntatore. La costante `NULL` è definita in `<stddef.h>`. D'ora in poi useremo `NULL`.

Condizioni come

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* memoria disponibile */
    e
```

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

chiariscono molti risvolti importanti dell'aritmetica dei puntatori. Innanzitutto, in alcune circostanze i puntatori possono essere confrontati. Se `p` e `q` puntano a elementi dello stesso vettore, relazioni quali `==`, `!=`, `<`, `>` e così via funzionano correttamente. Per esempio,

```
p < q
```

è vera se `p` punta a un elemento precedente a quello indicato da `q`. Qualunque puntatore può essere confrontato con zero per stabilirne l'egualanza o la disegualanza. Tuttavia, il comportamento del codice è indefinito se si eseguono operazioni aritmetiche o di confronto fra puntatori che non puntino a elementi dello stesso vettore. (Si dà un'eccezione: l'indirizzo del primo elemento dopo la fine di un vettore può essere usato nell'aritmetica dei puntatori.)

Secondariamente, abbiamo già constatato che un puntatore e un intero si possono sommare o sottrarre. L'istruzione

```
p + n
```

denota l'indirizzo dell'`n`-esimo oggetto dopo quello a cui punta attualmente `p`. Questo è vero a prescindere dalla natura dell'oggetto cui `p` punta; `n` è ridimensionato in relazione alla dimensione degli oggetti puntati da `p`, determinata dalla dichiarazione di `p`. Se un intero è di quattro byte, per esempio, e `p` punta a un intero, `n` sarà moltiplicato per quattro.

È valida anche la sottrazione fra puntatori: se `p` e `q` puntano a elementi dello stesso vettore, e `p < q`, allora `q - p + 1` è il numero di elementi da `p` a `q`, estremi inclusi.

Ciò consente di scrivere un'altra versione di `strlen`:

```
/* strlen: restituisce la lunghezza della stringa s */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

Nella sua dichiarazione, `p` è inizializzato a `s`, ovvero punta al primo carattere della stringa. Nel ciclo `while`, ogni carattere viene esaminato a turno finché non compaia lo `'\0'` finale. Poiché `p` punta a caratteri, `p++` fa avanzare ogni volta `p` al carattere successivo, e `p - s` dà il numero totale di caratteri in corrispondenza dei quali `p` è stato spostato in avanti, cioè la lunghezza della stringa. (La stringa potrebbe essere troppo lunga per essere contenuta in un intero; l'intestazione `<stddef.h>` definisce un tipo `ptrdiff_t` la cui dimensione è sufficiente a contenere la differenza con segno tra i valori di due puntatori. Tuttavia, volendo essere cau-

ti, useremmo `size_t` per il tipo restituito da `strlen`, in conformità alla versione della libreria standard; `size_t` è il tipo intero senza segno restituito dall'operatore `sizeof`.)

L'aritmetica dei puntatori è coerente: se avessimo avuto a che fare con variabili di tipo `float`, che occupano più spazio di quelle di tipo `char`, e se `p` fosse un puntatore a `float`, `p++` avanzerebbe al `float` successivo. A questo punto potremmo scrivere un'ulteriore versione di `alloc` che tratti `float` invece che `char`, semplicemente cambiando `char` in `float` all'interno di `alloc` e `afree`. Tutte le manipolazioni dei puntatori tengono conto automaticamente della grandezza dell'oggetto a cui puntano.

Le operazioni valide con i puntatori sono assegnamenti fra puntatori dello stesso tipo, somme o sottrazioni di un puntatore e un intero, sottrazioni o confronti di due puntatori a elementi dello stesso vettore, e assegnamenti o confronti con zero. Forme di aritmetica dei puntatori che esulino da questi casi sono illegali. Somme, moltiplicazioni, divisioni, scorimenti, operazioni sui singoli bit tramite maschere non sono operazioni ammesse. Parimenti, non si può sommare un valore `float` o `double` a un puntatore; a eccezione di `void *`, non è neppure ammesso l'assegnamento fra puntatori di tipo diverso, a meno di non forzare la conversione tramite il costrutto `cast`.

## 5.5 Puntatori a caratteri e funzioni

Una costante stringa, scritta come

`"Sono una stringa"`

è un vettore di caratteri. Nella rappresentazione interna, il vettore termina con il carattere nullo '\0' così che i programmi possano individuarne la fine. La lunghezza occupata in memoria è dunque di un carattere in più di quelli tra virgolette.

L'uso più comune delle costanti stringa è come argomenti delle funzioni, per esempio

```
printf("ciao, mondo\n");
```

Quando una simile stringa appare in un programma, l'accesso ad essa è mediato da un puntatore; `printf` riceve un puntatore all'inizio del vettore di caratteri. Pertanto, si accede al primo elemento di una costante stringa tramite un puntatore.

Una costante stringa non deve essere necessariamente argomento di una funzione. Se `pmessage` è dichiarato come

```
char *pmessage;
```

l'istruzione

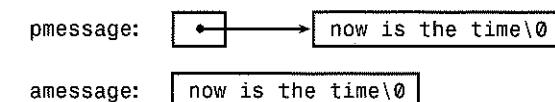
```
pmessage = "now is the time";
```

assegna a `pmessage` un puntatore al vettore di caratteri. Non si tratta della copia di una stringa; nell'operazione sono coinvolti solo puntatori. Il C non fornisce alcun operatore per trattare una stringa di caratteri come unità singola.

Esiste una differenza importante tra queste definizioni:

```
char amessage[] = "now is the time"; /* un vettore */
char *pmessage = "now is the time"; /* un puntatore */
```

`amessage` è un vettore, grande quanto basta per contenere la sequenza di caratteri (seguita da '\0') che lo inizializza. Singoli caratteri del vettore possono essere cambiati, ma `amessage` farà riferimento sempre alla stessa zona di memoria. D'altra parte, `pmessage` è un puntatore, inizializzato per puntare a una costante stringa; esso potrà essere modificato per denotare qualcosa' altro, ma il risultato è indefinito se si tenta di cambiare i contenuti della stringa.



Approfondiremo ulteriori aspetti dei puntatori e dei vettori esaminando alcune possibili implementazioni di due utili funzioni della libreria standard. La prima è `strcpy(s, t)`, che copia la stringa `t` nella stringa `s`. Sarebbe comodo affermare semplicemente `s=t`, ma otterremmo la copia del puntatore, non dei caratteri. Per duplicare i caratteri occorre un ciclo. La versione che si basa sui vettori è:

```
/* strcpy: copia t in s; versione vettoriale */
void strcpy(char *s, char *t)
{
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

Ecco invece una versione di `strcpy` con puntatori:

```
/* strcpy: copia t in s; versione 1 con i puntatori */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Poiché gli argomenti sono passati per valore, `strcpy` può usare i parametri `s` e `t` come ritiene più opportuno. In questo esempio si tratta di puntatori adeguatamente inizializzati, che vengono fatti scorrere lungo i vettori un carattere per volta, fino a quando il carattere '\0' di chiusura di `t` sia stato copiato in `s`.

Nella realtà, `strcpy` non sarebbe scritta come è stata presentata. Un programmatore esperto del C preferirebbe

```
/* strcpy: copia t in s; versione 2 con i puntatori */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

L'incremento di `s` e `t` ha luogo adesso nella condizione di terminazione del ciclo. Il valore di `*t++` è il carattere a cui `t` puntava prima del suo incremento; il suffisso `++` non modifica `t` finché questo carattere non sia stato trattato. Allo stesso modo, il carattere è memorizzato nella locazione puntata da `s` prima dell'incremento. Questo è anche il carattere che viene messo a confronto con `'\0'` per controllare il ciclo. Di conseguenza, i caratteri sono copiati da `t` a `s`, fino allo `'\0'` di chiusura compreso.

Si noti che il confronto con `'\0'` è ridondante, dato che ciò che interessa è che l'espressione sia zero. È quindi probabile che la funzione sarebbe scritta così:

```
/* strcpy: copia t in s; versione 3 con i puntatori */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Sebbene a una prima lettura questo codice possa risultare ermetico, il vantaggio di notazione è ragguardevole, tanto da rendere auspicabile una sicura padronanza di simili usi idiomatici del linguaggio: capita spesso di imbattersi in passi di questa natura nei programmi in C.

Nella libreria standard, la funzione `strcpy` (`<string.h>`) restituisce come suo valore la stringa copiata.

La seconda funzione che sottoponiamo ad analisi è `strcmp(s, t)`, che confronta le stringhe di caratteri `s` e `t`, e genera un risultato negativo, zero o positivo se `s` è lessicograficamente minore di, uguale a o maggiore di `t`. Il risultato deriva dalla sottrazione fra caratteri corrispondenti nella prima posizione in cui `s` e `t` differiscono.

```
/* strcmp: restituisce <0 se s<t, 0 se s==t, >0 se s>t */
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];
}
```

La versione con i puntatori di `strcmp` è:

```
/* strcmp: restituisce <0 se s<t, 0 se s==t, >0 se s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

Dal momento che `++` e `--` hanno senso sia come prefissi che come suffissi, sono possibili altre combinazioni di `*` con `++` e `--`, benché più rare. Per esempio,

`*--p`

decrementa `p` prima di accedere al carattere puntato da `p`. In realtà, la coppia di espressioni

```
*p++ = val; /* impila val sulla pila */
val = *--p; /* estrae la cima della pila, memorizzandola in val */
```

mostra le locuzioni più usate per impilare ed estrarre elementi su e da una pila; si veda il Paragrafo 4.3.

L'intestazione `<string.h>` contiene le dichiarazioni delle funzioni citate in questo paragrafo, oltre a una quantità di funzioni della libreria standard per il trattamento delle stringhe.

**Esercizio 5.3** Si scriva una versione comprensiva di puntatori della funzione `strcat` vista nel Capitolo 2; `strcat(s, t)` copia la stringa `t` al termine della stringa `s`.

**Esercizio 5.4** Si scriva una funzione `strend(s, t)`, che dia per risultato 1 se la stringa `t` coincide con una porzione finale della stringa `s`, e zero altrimenti.

**Esercizio 5.5** Si scrivano versioni delle funzioni della libreria `strncpy`, `strncat` e `strncmp`, che agiscano al massimo sui primi `n` caratteri dei loro argomenti di tipo stringa. Per esempio, `strncpy(s, t, n)` copia non più di `n` caratteri di `t` in `s`. Le descrizioni complete si trovano nell'Appendice B.

**Esercizio 5.6** Si riformulino programmi ed esercizi di capitoli precedenti con puntatori in luogo degli indici dei vettori. Fra gli spunti interessanti vi sono `getline` (Capitoli 1 e 4), `atoi`, `itoa`, e rispettive varianti (Capitoli 2, 3 e 4), `reverse` (Capitolo 3), `strindex` e `getop` (Capitolo 4).

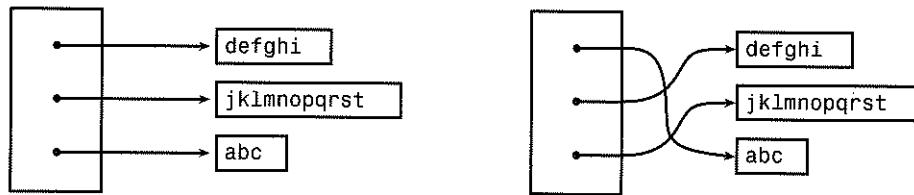
## 5.6 Vettori di puntatori; puntatori a puntatori

I puntatori possono essere memorizzati in vettori, dal momento che sono essi stessi delle variabili. Spieghiamo il concetto con un programma che ordina alfabeticamente una serie di righe di testo, una versione semplificata del programma UNIX `sort`.

Nel Capitolo 3 abbiamo presentato il metodo Shell per ordinare un vettore di interi, e nel Capitolo 4 l'abbiamo migliorato con l'algoritmo quicksort. Si potrebbe ricorrere anco-

ra all'uso di questi algoritmi, se non che abbiamo ora a che fare con righe di testo di diversa lunghezza che, a differenza degli interi, non è possibile confrontare o spostare in una singola operazione. Dobbiamo trovare una rappresentazione dei dati che permetta di trattare con efficienza e agilità righe di testo a lunghezza variabile.

Ed è qui che entra in gioco il vettore di puntatori. Se le righe di testo da ordinare sono memorizzate consecutivamente in un lungo vettore di caratteri, si potrà accedere a ogni riga tramite un puntatore al suo primo carattere. Gli stessi puntatori possono essere memorizzati in un vettore. Si ottiene il confronto tra due righe passandone i relativi puntatori a `strcmp`. Quando due righe fuori posto vanno scambiate, sono i puntatori a mutare collocazione nel vettore di puntatori e non le righe medesime.



Si elimina così il doppio problema legato allo spostamento fisico delle righe: la complicata gestione della memoria e la penalizzazione delle prestazioni.

Il processo di ordinamento segue tre fasi:

- leggi tutte le righe in ingresso*
- mettile in ordine*
- visualizzale nell'ordine corretto*

Come sempre, è bene che il programma sia strutturato in funzioni corrispondenti a questa divisione naturale, con la funzione `main` preposta al controllo delle altre. Rinviamo a dopo l'analisi della fase di ordinamento e concentriamoci sulla struttura dei dati e sulla gestione dei dati in ingresso e uscita.

Il modulo per la lettura deve mettere da parte i caratteri di ogni riga, e costruire un vettore di puntatori alle righe. Dovrà anche contare il numero di righe in ingresso, visto che l'informazione è necessaria per ordinarle e visualizzarle. Poiché questa funzione si limita a trattare un numero finito di righe, può restituire un segnale d'errore, diciamo -1, se la quantità dei dati in ingresso è eccessiva.

La funzione preposta all'uscita dei dati deve solo visualizzare le righe nell'ordine in cui appaiono nel vettore di puntatori.

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

#define MAXLINES 5000 /* numero massimo di righe ordinabili */

char *lineptr[MAXLINES]; /* puntatori alle righe del testo */
```

```
int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(char *lineptr[], int left, int right);

/* ordina le righe in ingresso */
main()
{
    int nlines; /* numero di righe in ingresso lette */

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("errore: troppi dati in ingresso da ordinare\n");
        return 1;
    }
}

#define MAXLEN 1000 /* massima lunghezza di una riga in ingresso */
int getline(char *, int);
char *alloc(int);

/* readlines: legge le righe in ingresso */
int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
    char *p, line[MAXLEN];

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || (p = alloc(len)) == NULL)
            return -1;
        else {
            line[len-1] = '\0'; /* cancella il carattere newline */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}

/* writelines: scrive le righe in uscita */
void writelines(char *lineptr[], int nlines)
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

La funzione `getline` si trova nel Paragrafo 1.9.

La novità di spicco è la dichiarazione di `lineptr`:

```
char *lineptr[MAXLINES];
```

che afferma che `lineptr` è un vettore di `MAXLINES` elementi, ognuno dei quali è un puntatore a carattere. In altre parole, `lineptr[i]` è un puntatore a carattere, e `*lineptr[i]` è il carattere cui esso punta, il primo carattere della `i`-esima riga di testo memorizzata.

Dato che `lineptr` è a sua volta il nome di un vettore, può essere trattato come un puntatore come già visto in precedenza, e `writelines` può essere riscritta così:

```
/* writelines: scrive le righe in uscita */
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}
```

Inizialmente `*lineptr` punta alla prima riga; ogni incremento lo fa avanzare al puntatore della riga seguente, mentre `nlines` decresce di pari passo.

Avendo sotto controllo i dati in ingresso e uscita, possiamo passare all'ordinamento. Il programma quicksort del Capitolo 4 richiede aggiustamenti di scarsa entità: le dichiarazioni devono essere modificate, e l'operazione di confronto deve essere effettuata chiamando `strcmp`. L'algoritmo rimane identico, un buon motivo per ritenere che continui a funzionare.

```
/* qsort: ordina v[left]...v[right] in ordine crescente */
void qsort(char *v[], int left, int right)
{
    int i, last;
    void swap(char *v[], int i, int j);

    if (left >= right) /* non fa nulla se il vettore contiene */
        return;           /* meno di due elementi */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

La funzione `swap`, analogamente, esige solo piccoli cambiamenti:

```
/* swap: scambia di posto v[i] e v[j] */
void swap(char *v[], int i, int j)
{
    char *temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
```

```
    v[j] = temp;
}
```

Poiché ogni singolo elemento di `v` (alias `lineptr`) è un puntatore di caratteri, anche `temp` deve esserlo, cosicché è possibile copiare l'uno nell'altro.

**Esercizio 5.7** Si riscriva `readlines` in modo da memorizzare le righe in un vettore fornito da `main`, invece di chiamare `alloc` allo stesso scopo. Quanto è più veloce il programma?

## 5.7 Vettori multidimensionali

Il C dispone anche di vettori rettangolari a più dimensioni, che in realtà sono molto meno usati dei vettori di puntatori. In questo paragrafo accenneremo ad alcune loro proprietà.

Si consideri il problema di convertire una data, dal giorno del mese al giorno dell'anno e viceversa. Per esempio, il 1° marzo è il sessantesimo giorno di un anno regolare, e il sessantunesimo di un anno bisestile (nel codice, “leap year”). Definiamo due funzioni che effettuino le conversioni: `day_of_year` trasforma il giorno del mese nel giorno dell'anno, mentre `month_day` converte il giorno dell'anno nel giorno del mese, indicando anche il mese. Dato che quest'ultima funzione calcola due valori, gli argomenti `mese` e `giorno` saranno puntatori:

```
month_day(1988, 60, &m, &d)
```

imposta `m` a 2 e `d` a 29 (29 febbraio).

Queste funzioni hanno entrambe bisogno della stessa informazione, una tabella con il numero di giorni di ogni mese (“Trenta giorni ha settembre...”). Visto che il numero di giorni cambia se l'anno è bisestile, è più facile trattare i due casi separatamente in due righe di un vettore bidimensionale che tenere conto di come si comporta febbraio nel corso del calcolo. Il vettore e le funzioni per compiere le trasformazioni sono riportati di seguito:

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 31}, // non bisestile
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 31} // bisestile
};

/* day_of_year: imposta giorno dell'anno dal mese & giorno */
int day_of_year(int year, int month, int day)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}

/* month_day: imposta mese & giorno dal giorno dell'anno */
void month_day(int year, int yearday, int *pmmonth, int *pday)
{
    int i, leap;
```

```

leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
for (i = 1; yearday > daytab[leap][i]; i++)
    yearday -= daytab[leap][i];
*pmonth = i;
*pday = yearday;
}

```

Si rammenti che il valore aritmetico di un'espressione logica, come quello assegnato a `leap`, è o zero (falso) o uno (vero), e in quanto tale può essere usato come indice del vettore `daytab`.

Perché entrambe possano usarlo, il vettore `daytab` deve essere esterno sia a `day_of_year` che a `month_day`. Lo si è dichiarato `char`, per illustrare un impiego legittimo di quest'ultimo tipo per memorizzare piccoli interi che non siano caratteri.

La variabile `daytab` è il primo vettore bidimensionale che abbiamo incontrato. Nel C, un vettore bidimensionale è in realtà un vettore a una sola dimensione, ogni elemento del quale è un vettore. Gli indici si adoperano quindi così:

```

daytab[i][j] /* [riga][colonna] */
invece di
daytab[i,j] /* ERRATO */

```

Al di là di questa distinzione formale, un vettore bidimensionale può essere trattato in buona sostanza come avviene in altri linguaggi. Gli elementi sono memorizzati per righe, cosicché l'indice di destra, o colonna, è quello che cambia più velocemente mentre si accede agli elementi in ordine di memorizzazione.

Un vettore viene inizializzato da una lista di inizializzatori tra parentesi graffe; ogni riga di un vettore bidimensionale è inizializzata da una sotto-lista corrispondente. Abbiamo fatto partire il vettore `daytab` da una colonna di zeri in modo che i numeri relativi al mese fossero compresi tra 1 e 12 invece che tra 0 e 11. Visto che non ci sono problemi di spazio, ciò risulta più chiaro rispetto all'adattamento degli indici.

Se si vuole passare un vettore bidimensionale a una funzione, la dichiarazione del parametro nella funzione deve includere il numero di colonne; il numero delle righe è irrilevante, dal momento che l'oggetto passato è, come prima, un puntatore a un vettore, in cui ogni elemento è esso stesso un vettore. Nel caso dell'esempio precedente, si passa un puntatore a oggetti che siano vettori di 13 interi. Pertanto se il vettore `daytab` dovesse essere passato a una funzione `f`, la dichiarazione di `f` sarebbe

```
f(int daytab[2][13]) { ... }
```

E potrebbe anche essere

```
f(int daytab[][][13]) { ... }
```

vista l'irrilevanza del numero delle righe; oppure potrebbe essere

```
f(int (*daytab)[13]) { ... }
```

che dice che il parametro è un puntatore a un vettore di 13 interi. Le parentesi tonde si rendono necessarie dato che le parentesi quadre [] hanno diritto di precedenza su \*.

Priva di parentesi, la dichiarazione

```
int *daytab[13]
```

è un vettore di 13 puntatori a interi. Più in generale, è solo la prima dimensione (indice) di un vettore a essere libera; tutte le altre devono essere specificate.

Nel Paragrafo 5.12 vi è un ulteriore approfondimento delle dichiarazioni complesse.

**Esercizio 5.8** Le funzioni `day_of_year` e `month_day` non prevedono il trattamento degli errori. Il lettore ponga rimedio a questa mancanza.

## 5.8 Inizializzazione dei vettori di puntatori

Si consideri il problema della realizzazione di una funzione `month_name(n)`, che restituisca un puntatore a una stringa di caratteri contenente il nome dell'*n*-esimo mese. Questa è un'applicazione ideale per un vettore interno `static`. La funzione `month_name` contiene un vettore locale di stringhe di caratteri, e, quando è chiamata, restituisce un puntatore a quella opportuna. Questo paragrafo mostra come inizializzare quel vettore di nomi.

La sintassi è simile alle inizializzazioni già incontrate:

```

/* month_name: restituisce il nome del mese n-esimo */
char *month_name(int n)
{
    static char *name[] = {
        "mese inesistente",
        "gennaio", "febbraio", "marzo",
        "aprile", "maggio", "giugno",
        "luglio", "agosto", "settembre",
        "ottobre", "novembre", "dicembre"
    };
    return (n < 1 || n > 12) ? name[0] : name[n];
}

```

La dichiarazione di `name`, che è un vettore di puntatori di caratteri, è la medesima di `lineptr` nell'esempio dell'ordinamento. L'inizializzatore è una lista di stringhe di caratteri; a ognuna è attribuita la posizione corrispondente all'interno del vettore. I caratteri della *i*-esima stringa sono collocati da qualche parte, mentre un puntatore al primo carattere della stringa è memorizzato in `name[i]`. Poiché la grandezza del vettore `name` non è precisata, il compilatore la deduce contando gli inizializzatori.

## 5.9 Puntatori e vettori multidimensionali a confronto

I neofiti del C rischiano talvolta di confondersi sulla differenza tra un vettore bidimensionale e un vettore di puntatori, quale per esempio il vettore `name` appena visto. Date le definizioni

```

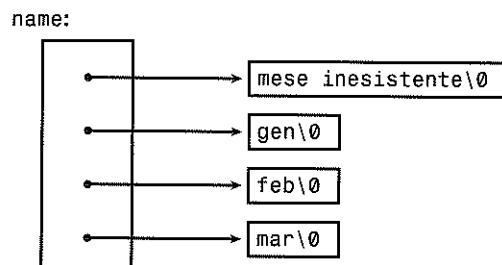
int a[10][20];
int *b[10];

```

`a[3][4]` e `b[3][4]` sono entrambi, dal punto di vista sintattico, riferimenti corretti a un singolo intero. Ma `a` è un vettore bidimensionale vero e proprio: sono state riservate 200 locazioni della grandezza di un intero, e l'elemento convenzionalmente indicato da `a[riga][colonna]` si trova in realtà nella posizione  $20 \times \text{riga} + \text{colonna}$ . Quanto a `b`, invece, la definizione stanzia solo 10 puntatori e non li inizializza; l'inizializzazione deve avvenire esplicitamente, staticamente o tramite del codice. Assumendo che ogni elemento di `b` punti realmente a un vettore di venti elementi, si avrebbero 200 interi allocati, più dieci locazioni per i puntatori. Il vantaggio decisivo del vettore di puntatori è che consente diverse lunghezze per le righe. Ciò significa che non tutti gli elementi di `b` devono puntare a un vettore di venti elementi; qualcuno di essi può puntare a due elementi, altri a cinquanta, e altri ancora a nessun elemento.

Sebbene finora l'oggetto della trattazione siano stati gli interi, l'uso di gran lunga più frequente dei vettori di puntatori è la memorizzazione di stringhe di caratteri di diverse lunghezze, come nella funzione `month_name`. Si confrontino la dichiarazione e la rappresentazione grafica seguente di un vettore di puntatori

```
char *name[] = { "mese inesistente", "gen", "feb", "mar" };
```



con quelle relative a un vettore bidimensionale:

```
char aname[][20] = { "mese inesistente", "gen", "feb", "mar" };
```

aname:

	mese inesistente\0	gen\0	feb\0	mar\0
0		20	40	60

**Esercizio 5.9** Il lettore riscriva le funzioni `day_of_year` e `month_day` avvalendosi di puntatori invece che di vettori e indici.

## 5.10 Argomenti dalla riga di comando

In ambienti che permettono la stesura di programmi in C, c'è la possibilità di passare argomenti o parametri dalla riga di comando a un programma al momento dell'esecuzione. La funzione `main` riceve due argomenti. Il primo (chiamato per convenzione `argc`, da *argument count*) è il numero di argomenti presenti nella riga di comando; il secondo (`argv`, da *argument vector*) è un puntatore a un vettore di stringhe di caratteri che contengono gli argo-

menti, uno per stringa. Si adoperano abitualmente svariati livelli di puntatori per intervenire su queste stringhe di caratteri.

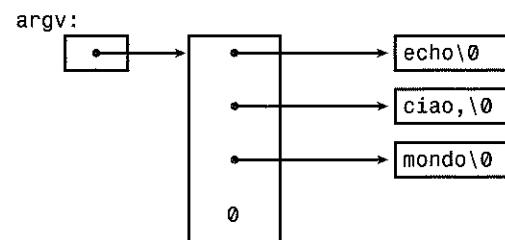
L'esemplificazione più semplice è data dal programma `echo`, che visualizza gli argomenti ricevuti dalla riga di comando, inframmezzati da spazi, su una riga singola. In pratica, il comando

```
echo ciao, mondo
```

produce

```
ciao, mondo
```

Per convenzione, `argv[0]` è il nome con cui il programma è stato invocato, quindi `argc` vale sempre almeno 1; quando è proprio uguale a 1, non si danno argomenti nella riga di comando dopo il nome del programma. Nell'esempio precedente, `argc` è 3, e `argv[0]`, `argv[1]`, `argv[2]` sono rispettivamente "echo", "ciao," e "mondo". Il primo argomento faticativo è `argv[1]` e l'ultimo è `argv[argc-1]`; inoltre, lo standard prescrive che `argv[argc]` sia un puntatore nullo.



La prima versione di `echo` tratta `argv` come un vettore di puntatori di caratteri:

```
#include <stdio.h>

/* eco degli argomenti dalla riga di comando; prima versione */
main(int argc, char *argv[])
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s %s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

Dal momento che `argv` è un puntatore a un vettore di puntatori, possiamo agire sul puntatore invece di usare gli indici. Questa variante si fonda su un opportuno incremento di `argv`, che è un puntatore a un puntatore a carattere, mentre `argc` è progressivamente decrementato fino a zero:

```
#include <stdio.h>

/* eco degli argomenti dalla riga di comando; seconda versione */
main(int argc, char *argv[])
{
    int i;
```

```

{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}

```

Posto che `argv` è un puntatore all'inizio del vettore delle stringhe contenenti gli argomenti, dopo l'incremento di 1 (`*++argv`) esso punta all'originario `argv[1]` piuttosto che ad `argv[0]`. Ogni incremento successivo lo sposta all'argomento seguente; `*argv` è dunque il puntatore a tale argomento. Allo stesso tempo, `argc` diminuisce; quando diventa zero, non restano argomenti da visualizzare.

In alternativa, potremmo riscrivere l'istruzione `printf` come

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

il che mostra incidentalmente che anche il primo argomento di `printf` può essere un'espressione.

Come secondo esempio, apporteremo qualche miglioria al programma di ricerca di un pattern in un testo visto nel Paragrafo 4.1. Come si ricorderà, avevamo cablato nel cuore del programma la stringa da cercare, una soluzione che senza dubbio lasciava a desiderare. Cogliendo il suggerimento offerto dal programma `grep` di UNIX, modifichiamo il codice affinché il pattern da trovare sia specificato dal primo argomento nella linea di comando.

```

#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: visualizza le righe in ingresso che contengono il pattern */
/* specificato dal primo argomento della riga di comando */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;

    if (argc != 2)
        printf("Uso: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
            }
    return found;
}

```

La funzione della libreria standard `strstr(s,t)` restituisce un puntatore alla prima occorrenza della stringa `t` nella stringa `s`, ovvero `NULL` se `t` non compare affatto in `s`. La dichiarazione si trova in `<string.h>`.

Il modello può essere ora elaborato per esemplificare altri costrutti con puntatori. Si supponga di voler consentire due argomenti facoltativi. Il primo richiede di “visualizzare tutte le righe *eccetto* quelle che contengono il pattern,” il secondo di “anteporre a ogni riga visualizzata il proprio numero di riga.”

Una convenzione comune per i programmi in C ospitati da ambienti UNIX è che un argomento preceduto da un segno meno introduce un parametro facoltativo, detto *flag*. Se sceglieremo `-x` (in riferimento a “eccetto”) come segnale dell'inversione, e `-n` (in riferimento a “numero”) per richiedere la numerazione delle righe, il comando

```
find -x -n pattern
```

visualizzerà ogni riga in ingresso che non contenga il pattern, preceduta dal suo numero di riga.

Gli argomenti facoltativi dovrebbero essere consentiti in qualunque ordine, mentre il resto del programma dovrebbe essere indipendente dal numero di argomenti presenti. E ancora, sarebbe vantaggiosa per gli utenti la possibilità di combinare argomenti facoltativi, come in

```
find -nx pattern
```

Ecco il programma:

```

#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: visualizza le righe in ingresso che contengono il pattern */
/* specificato dal primo argomento della riga di comando */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    long lineno = 0;
    int c, except = 0, number = 0, found = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
                case 'x':
                    except = 1;
                    break;
                case 'n':
                    number = 1;
                    break;
                default:
                    printf("find: opzione illegale %c\n", c);
                    argc = 0;
                    found = -1;
                    break;
            }
    if (argc > 0)
        printf("Uso: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if ((number == 0) || (strstr(line, argv[0]) != NULL) != except)
                if (except)
                    printf("%ld: %s", lineno, line);
                else
                    printf("%s", line);
                lineno++;
}

```

```

if (argc != 1)
    printf("Uso: find -x -n pattern\n");
else
    while (getline(line, MAXLINE) > 0) {
        lineno++;
        if ((strstr(line, *argv) != NULL) != except) {
            if (number)
                printf("%d:", lineno);
            printf("%s", line);
            found++;
        }
    }
return found;
}

```

Nel primo ciclo `while`, `argc` è diminuito e `argv` aumentato prima dell'esame di un dato argomento facoltativo. Alla fine del ciclo, sempre che non ci siano errori, `argc` contiene il numero degli argomenti rimasti da elaborare e `argv` punta al primo di essi. Di conseguenza, `argc` dovrebbe essere 1 e `*argv` dovrebbe puntare al `pattern`. Si osservi come `++argv` sia un puntatore a una stringa contenente un argomento, e quindi `(++argv)[0]` è il suo primo carattere. (Una forma alternativa valida sarebbe `***+argv`.) Le parentesi sono necessarie; visto che `[]` ha diritto di precedenza su `*` e `++`, in loro assenza l'espressione sarebbe intesa come `++(argv[0])`. In effetti, la versione senza parentesi compare nel ciclo `while` interno, dove l'obiettivo è di scandire una data stringa contenente un argomento. In questo ciclo, l'espressione `++argv[0]` incrementa il puntatore `argv[0]`!<sup>(1)</sup>

È raro che si usino espressioni con puntatori più complicate di queste; se l'occasione si presentasse, la loro scomposizione in due o tre passi porterà a un codice più comprensibile.

**Esercizio 5.10** Si scriva un programma `expr`, che valuti un'espressione in notazione polacca dalla riga di comando, dove ogni operatore od operando è un argomento separato. Per esempio,

```

expr 2 3 4 + *
computa 2 × (3+4).

```

**Esercizio 5.11** Il lettore modifichi i programmi `entab` e `datab` (proposti sotto forma di esercizi nel Capitolo 1) cosicché accettino come argomenti una lista di tabulazioni. In mancanza di argomenti, si imposti un valore di default, come nella prima versione dei programmi.

1. N.d.R. Nell'errata corrige al testo, pubblicata elettronicamente, si legge: "Il programma `find` [quello appena visto] incrementa `argv[0]`, il che non è esplicitamente proibito [dallo standard ANSI], ma non è neppure esplicitamente consentito."

**Esercizio 5.12** Si estendano `entab` e `datab` in modo che accettino l'abbreviazione  
`entab -m +n`

che specifica una tabulazione ogni `n` colonne, a partire dalla colonna `m`. Si adotti un comportamento di default conveniente per l'utente.

**Esercizio 5.13** Si scriva il programma `tail`, che visualizzi le ultime `n` righe del proprio testo in ingresso. Come impostazione predefinita, si ponga `n` uguale a 10. Si preveda la possibilità di alterare `n` tramite un argomento facoltativo, di modo che

```
tail -n
```

visualizzi le ultime `n` righe. Il programma dovrebbe comportarsi bene, a prescindere dalla ragionevolezza del valore di `n` o del testo in ingresso. Si scriva il programma in modo tale che la memoria disponibile sia sfruttata al massimo; le righe dovrebbero essere memorizzate come nel programma di ordinamento del Paragrafo 5.6, e non in un vettore bidimensionale di dimensione fissa.

## 5.11 Puntatori a funzioni

In C, una funzione non è di per sé una variabile, ma è possibile definire puntatori a funzioni, che si possono poi assegnare, memorizzare in un vettore, passare a funzioni, restituire tramite funzioni, e così via. Illustreremo questo aspetto adottando l'algoritmo di ordinamento già mostrato in questo capitolo in modo che, in presenza dell'argomento facoltativo `-n`, ordini le righe in ingresso numericamente e non secondo l'ordine alfabetico (lessicograficamente).

Una procedura d'ordinamento è spesso costituita da tre parti: un confronto che determina la relazione d'ordine fra una coppia arbitraria d'oggetti; uno scambio che inverta i membri della coppia; l'algoritmo d'ordinamento in senso proprio, che esegue confronti e scambi fino a che tutti gli oggetti coinvolti siano nell'ordine corretto. L'algoritmo di ordinamento è indipendente dalle procedure di confronto e scambio, il che rende possibile ordinare secondo criteri diversi modificando tali procedure. È questo l'approccio adottato nella seguente nuova versione del programma.

Il confronto lessicografico fra due righe è eseguito, come già prima, da `strcmp`; avremo anche bisogno di una funzione `numcmp` per confrontare due righe in base al loro valore numerico e restituire valori analoghi a quelli forniti da `strcmp`. Le dichiarazioni di queste funzioni precedono `main`, e un puntatore a quella appropriata è passato a `qsort`. Abbiamo lasciato da parte la questione della gestione degli errori negli argomenti in ingresso, al fine di concentrare l'attenzione sui punti salienti del problema.

```

#include <stdio.h>
#include <string.h>

#define MAXLINES 5000      /* numero massimo di righe ordinabili */
char *lineptr[MAXLINES]; /* puntatori alle righe del testo */

```

```

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(void *lineptr[], int left, int right,
           int (*comp)(void *, void *));
int numcmp(char *, char *);

/* ordina le righe in ingresso */
main(int argc, char *argv[])
{
    int nlines;          /* numero di righe in ingresso lette */
    int numeric = 0;     /* vale 1 se l'ordinamento e' numerico */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void**) lineptr, 0, nlines-1,
               (int (*)(void*,void*))(numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("troppi dati in ingresso da ordinare\n");
        return 1;
    }
}

```

Nella chiamata a `qsort`, `numcmp` e `strcmp` sono indirizzi di funzioni; come nel caso dei nomi dei vettori, non è necessario premettere l'operatore `&`.

Abbiamo progettato `qsort` in modo che possa trattare dati generici, e non solo stringhe di caratteri. Il prototipo della funzione specifica che `qsort` si aspetta un vettore di puntatori, due interi e una funzione con due puntatori come argomenti. Il tipo `void *` denota un puntatore a oggetti di tipo arbitrario: si può sempre convertire un puntatore al tipo `void *` tramite `cast`, e viceversa, senza che vi sia alcuna perdita d'informazione; ciò permette di invocare `qsort` forzando il tipo degli argomenti a essere `void *`. L'elaborata conversione nella chiamata a `qsort` forza il tipo degli argomenti della funzione di confronto: la cosa non ha alcun effetto sulla reale rappresentazione dei dati, ma serve a rassicurare il compilatore.

```

/* qsort: ordina v[left]...v[right] in ordine crescente */
void qsort(void *v[], int left, int right,
           int (*comp)(void *, void *))
{
    int i, last;
    void swap(void *v[], int, int);

    if (left >= right)      /* non fa nulla se il vettore contiene */
        return;                /* meno di due elementi */

    swap(v, left, (left + right)/2);
    last = left;

```

```

        for (i = left+1; i <= right; i++)
            if ((*comp)(v[i], v[left]) < 0)
                swap(v, ++last, i);
        swap(v, left, last);
        qsort(v, left, last-1, comp);
        qsort(v, last+1, right, comp);
    }

```

È opportuno studiare con cura le dichiarazioni in questo esempio. Il quarto parametro di `qsort`,

`int (*comp)(void *, void *)`

stipula che `comp` sia un puntatore a una funzione con due argomenti di tipo `void *`, e che restituisca un intero. L'uso di `comp` nella riga

`if ((*comp)(v[i], v[left]) < 0)`

è coerente con la dichiarazione: `comp` è un puntatore a una funzione, `*comp` è la funzione e `(*comp)(v[i], v[left])`

è una chiamata alla funzione `*comp`. Si noti che le parentesi sono necessarie; senza di esse la dichiarazione

`int *comp(void *, void *) /* ERRATO */`

asserisce che `comp` è una funzione che restituisce un puntatore a un intero, che è cosa ben diversa da quanto visto sopra.

Abbiamo già illustrato come `strcmp` confronti due stringhe. Passiamo ora a `numcmp`, che confronta due stringhe sulla base dei valori numerici da esse estratti tramite chiamata ad `atof`:

```

#include <stdlib.h>

/* numcmp: confronta s1 e s2 numericamente */
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}

```

La funzione `swap`, che scambia due puntatori, è identica alla sua ultima versione omonima già vista in questo capitolo, eccetto per l'uso del tipo più generale `void *` in luogo di `char *`.

```
void swap(void *v[], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Il programma di ordinamento può essere arricchito con una varietà di altre opzioni. Alcuni di questi emendamenti sono degli esercizi piuttosto impegnativi.<sup>(2)</sup>

**Esercizio 5.14** Si aggiunga al programma per l'ordinamento l'opzione `-r` (per *reverse*), che richiede l'ordinamento in ordine inverso (decrescente). L'opzione `-r` dovrà funzionare correttamente anche in presenza dell'opzione `-n`.

**Esercizio 5.15** Si preveda l'opzione `-f` per eliminare la distinzione fra maiuscole e minuscole (da “fold upper and lower case together”); per esempio, le lettere `a` e `A` risulterebbero uguali in presenza di questa opzione.

**Esercizio 5.16** Si aggiunga l'opzione `-d` per richiedere confronti solo fra lettere, numeri e spazi. (Il nome dell'opzione è giustificato dal fatto che, su molti sistemi, questi sono gli unici caratteri rilevanti per i nomi delle directory.) Ci si assicuri del corretto funzionamento congiunto delle opzioni `-d` e `-f`.

**Esercizio 5.17** Si amplii il programma fino a garantire la gestione di campi d'ordinamento diversi all'interno delle righe in ingresso, di modo che i singoli campi siano ordinabili sulla base di opzioni autonome.<sup>(3)</sup>

2. N.d.Rev. Riguardo al programma `qsort`, il lettore tenga presente i commenti seguenti, pubblicati elettronicamente come parte della errata corrigé. “Per molti aspetti, l'intera discussione [relativa a `qsort`] andrebbe rivista. Prima di tutto, `qsort` è una funzione della libreria standard ANSI/ISO C, per cui la stesura qui riportata andrebbe chiamata in modo diverso, in particolar modo alla luce del fatto che gli argomenti delle due versioni sono differenti. La funzione standard si aspetta un puntatore che funga da indirizzo base del vettore da ordinare, assieme a un indice che dia il numero degli elementi del vettore. L'esempio del testo, invece, sfrutta un puntatore base e due scostamenti.

Inoltre, il trattamento riservato all'argomento di `qsort` che specifica la funzione di confronto lascia a desiderare. La chiamata [che compare in `main`], con argomento

```
(int (*)(void*,void*))((numeric ? numcmp : strcmp)
```

oltre a essere complicata, rientra a malapena nel quadro dello standard. Sia `numcmp` che `strcmp` hanno argomenti `char *`, mentre l'espressione qui sopra forza i puntatori a queste funzioni nel tipo puntatore a una funzione con argomenti `void *`. Lo standard, in effetti, asserisce che `void *` e `char *` hanno la medesima rappresentazione, per cui il codice risulta perlomeno difendibile sul piano teorico e, nella pratica, girerà quasi certamente [su tutti i sistemi].”

3. N.d.Rev. L'indice analitico del volume originale americano è stato ordinato con l'opzione `-df` applicata ai lemmi, e l'opzione `-n` applicata ai numeri delle pagine.

## 5.12 Dichiarazioni complesse

Il C è a volte criticato per la sintassi delle dichiarazioni, specialmente quando esse coinvolgono puntatori alle funzioni. La filosofia che sta dietro alla sintassi del linguaggio è che le dichiarazioni e l'uso degli oggetti concordino: la cosa funziona bene nei casi più semplici, ma va a scapito della chiarezza al crescere della complessità. Ciò è dovuto a due fattori: non si possono leggere le dichiarazioni da sinistra a destra, e sono necessarie troppe parentesi. La differenza fra

```
int *f(); /* f: funzione che restituisce un puntatore a un intero */
```

e

```
int (*pf)(); /* pf: puntatore a una funzione che restituisce un intero */
```

illustra bene il problema: `*` è un operatore prefisso che deve dare precedenza a `()`, per cui le parentesi sono necessarie per indurre l'accorpamento corretto degli operandi.

Le dichiarazioni davvero complicate sono rare nella pratica, ma è importante essere in grado di comprenderle e, all'occorrenza, di impiegarle. Un modo per semplificare la stesura e la lettura delle dichiarazioni complesse è l'applicazione di `typedef`, uno strumento discusso nel Paragrafo 6.7. Come alternativa, presentiamo qui una coppia di programmi che convertono una dichiarazione corretta in C in una sua descrizione verbale, e viceversa. La descrizione verbale può essere letta da sinistra a destra.

Il primo programma della coppia, che chiameremo `dcl` (da “declarator”), è anche il più complesso: trasforma una dichiarazione C in una sua descrizione verbale, come negli esempi seguenti.

```
char **argv
    argv: puntatore a puntatore a char
int (*daytab)[13]
    daytab: puntatore a vettore[13] di int
int *daytab[13]
    daytab: vettore[13] di puntatore a int
void *comp()
    comp: funzione che restituisce puntatore a void
void (*comp)()
    comp: puntatore a funzione che restituisce void
char (*(*x())[5])
    x: funzione che restituisce puntatore a vettore[] di
        puntatore a funzione che restituisce char
char (*(*x[3])())[5]
    x: vettore[3] di puntatore a funzione che restituisce
        puntatore a vettore[5] di char
```

Il programma `dcl` si basa sulla grammatica formale che definisce un dichiaratore (*dich*) del linguaggio C; i dettagli sono nell'Appendice A, Paragrafo 8.5. Di seguito ne diamo una versione semplificata.

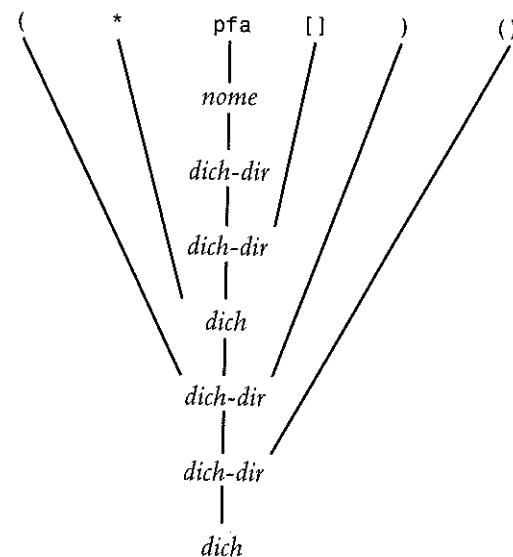
*dich:* uno o più \* facoltativi *dich-diretto*  
*dich-diretto:* *nome*  
*(dich)*  
*dich-diretto()*  
*dich-diretto[dimensione facoltativa]*

A parole, un *dich* è un *dich-diretto*, potenzialmente preceduto da uno o più \*. Un *dich-diretto*, a sua volta, è un *nome*, oppure un *dich* fra parentesi, oppure un *dich-diretto* seguito da parentesi, o infine un *dich-diretto* seguito da parentesi quadre, fra le quali si trova facoltativamente la specifica della dimensione.

Questa grammatica permette l'analisi sintattica automatica delle dichiarazioni. Si consideri, per esempio, il dichiaratore:

(\*pfa[]())

La sequenza pfa sarà identificata come un *nome*, e dunque come un *dich-diretto*; quindi pfa[] è anch'esso un *dich-diretto*. Ne segue che \*pfa[] è un *dich*, e (\*pfa[]) è un *dich-diretto*. Ma allora (\*pfa[]()) è un *dich-diretto*, e dunque un *dich*. L'analisi sintattica è bene illustrata da un albero come quello presentato di seguito, dove *dich-diretto* è abbreviato in *dich-dir*.



Il cuore del programma *dcl* è una coppia di funzioni, *dcl* e *dirdcl* (per “direct declarator”), che esegue l'analisi sintattica di una dichiarazione secondo questa grammatica. Poiché essa è definita ricorsivamente, le funzioni si chiamano a vicenda in modo ricorsivo man mano che riconoscono parti di una dichiarazione. Un programma di questo tipo è detto *analizzatore sintattico* (“parser”) ricorsivo discendente.

```

/* dcl: analizza sintatticamente un dichiaratore */
void dcl(void)
{
    int ns;

    for (ns = 0; gettoken() == '*' ; ) /* conta gli * */
        ns++;
    dirdcl();
    while (ns-- > 0)
        strcat(out, " puntatore a");
}

/* dirdcl: analizza sintatticamente un dichiaratore diretto */
void dirdcl(void)
{
    int type;

    if (tokentype == '(') {           /* ( dich ) */
        dcl();
        if (tokentype != ')')
            printf("errore: manca una )\n");
    } else if (tokentype == NAME) /* nome di variabile */
        strcpy(name, token);
    else
        printf("errore: era atteso nome o (dich)\n");
    while ((type=gettoken()) == PARENS || type == BRACKETS)
        if (type == PARENS)
            strcat(out, " funzione che restituisce");
        else {
            strcat(out, " vettore");
            strcat(out, token);
            strcat(out, " di");
        }
}
  
```

Il programma ha scopi didattici, e non è certo a prova di bomba. La funzione *dcl* soffre di alcune notevoli limitazioni: gestisce solo i tipi più semplici, come *char* e *int*; non è in grado di trattare i tipi degli argomenti delle funzioni, o gli attributi come *const*; gli spazi bianchi spuri sono fonte di difficoltà, come pure le dichiarazioni scorrette, a causa dell'assenza quasi totale della gestione degli errori. Le relative migliorie sono delegate agli esercizi.

Ecco la funzione principale e le variabili globali:

```

#include <stdio.h>
#include <string.h>
#include <cctype.h>

#define MAXTOKEN 100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);
  
```

```

int gettoken(void);
int tokentype;           /* tipo dell'ultimo token */
char token[MAXTOKEN];   /* ultimo token (una stringa) */
char name[MAXTOKEN];    /* nome dell'identificatore */
char datatype[MAXTOKEN]; /* tipo del dato = char, int, etc. */
char out[1000];          /* stringa in uscita */

main() /* trasforma dichiarazioni in descrizioni verbali */
{
    while (gettoken() != EOF) { /* il primo token della riga in */
        strcpy(datatype, token); /* ingresso e' il tipo del dato */
        out[0] = '\0';
        dcl(); /* analizza il resto della riga */
        if (tokentype != '\n')
            printf("errore sintattico\n");
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}

```

La funzione `gettoken` salta gli spazi e i caratteri di tabulazione, ed estrae il token successivo dai dati in ingresso. Un “token” è un nome, un coppia di parentesi, una coppia di parentesi quadre che racchiudono facoltativamente un numero, o un qualunque altro carattere singolo; per maggiori informazioni sul concetto di token, si veda l’Appendice A (A.2.1).

```

int gettoken(void) /* restituisce il prossimo token */
{
    int c, getch(void);
    void ungetch(int);
    char *p = token;

    while ((c = getch()) == ' ' || c == '\t')
        ;
    if (c == '(') {
        if ((c = getch()) == ')') {
            strcpy(token, "()");
            return tokentype = PARENS;
        } else {
            ungetch(c);
            return tokentype = '(';
        }
    } else if (c == '[') {
        for (*p++ = c; (*p++ = getch()) != ']'; )
            ;
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (isalpha(c)) {
        for (*p++ = c; isalnum(c = getch()); )
            *p++ = c;
        *p = '\0';
    }
}
```

```

        ungetch(c);
        return tokentype = NAME;
    } else
        return tokentype = c;
}

```

Le funzioni `getch` e `ungetch` sono state approfondite nel Capitolo 4.

Il programma complementare a quello appena visto, che chiameremo `undcl`, è più semplice, specie se non ci si preoccupa di limitare le parentesi ridondanti. Esso trasforma una descrizione verbale come “`x` è una funzione che restituisce un puntatore a un vettore di puntatori a funzioni che restituiscono un carattere”, che noi abbrevieremo in

`x () * [] * () char`

in un’espressione come

`char (*(*x())[]))()`

Il fatto di adottare una sintassi abbreviata in ingresso ci permette di riutilizzare la funzione `gettoken`. Si noti che `undcl` impiega le medesime variabili esterne di `dcl`.

```

/* undcl: trasforma descrizioni verbali in dichiarazioni */
main()
{
    int type;
    char temp[MAXTOKEN];

    while (gettoken() != EOF) {
        strcpy(out, token);
        while ((type = gettoken()) != '\n')
            if (type == PARENS || type == BRACKETS)
                strcat(out, token);
            else if (type == '*') {
                sprintf(temp, "(%*s)", out);
                strcpy(out, temp);
            } else if (type == NAME) {
                sprintf(temp, "%s %s", token, out);
                strcpy(out, temp);
            } else
                printf("input scorretto in %s\n", token);
    }
    return 0;
}

```

**Esercizio 5.18** Si faccia in modo che `dcl` gestisca opportunamente gli errori nei dati in ingresso.

**Esercizio 5.19** Si modifichi `undcl` in modo che non aggiunga parentesi ridondanti nelle dichiarazioni.

**Esercizio 5.20** Si estenda `dcl` in modo che riesca a gestire dichiarazioni con tipi degli argomenti delle funzioni, attributi come `const` e così via.

## Strutture

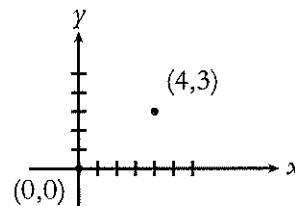
UNA STRUTTURA È UN INSIEME DI UNA O PIÙ VARIABILI, presumibilmente di tipo diverso, raggruppate sotto un nome comune per facilitarne la gestione. In alcuni linguaggi, in particolare il Pascal, le strutture prendono il nome di "record". Esse aiutano a organizzare dati complessi, e sono utili specialmente nel caso di programmi di grandi dimensioni, poiché consentono di trattare un gruppo di variabili come singola unità anziché entità separate.

Un esempio tipico di struttura è la busta paga: un impiegato è definito da un complesso di dati quali il nome, l'indirizzo, il codice di previdenza sociale, la retribuzione, e così via. Alcuni di questi dati potrebbero essere a loro volta strutture: un nome è formato da diverse voci, così come un indirizzo e la stessa retribuzione. Un altro esempio, più tipico del C, proviene dalla grafica: un punto è una coppia di coordinate, un rettangolo è determinato da una coppia di punti, e così via.

La principale innovazione dello standard ANSI è la definizione dell'assegnamento fra strutture; le strutture possono essere copiate, assegnate, passate alle funzioni e restituite da queste. Molti compilatori offrono queste funzionalità da lungo tempo, ma adesso le relative proprietà sono state formalizzate con precisione. Anche le strutture e i vettori automatici possono ora essere inizializzati.

## 6.1 Fondamenti delle strutture

Partiamo dalla stesura di alcune strutture pensate per applicazioni grafiche. L'oggetto base è un punto, per il quale ipotizziamo una coordinata  $x$  e una coordinata  $y$ , entrambe intere.



Le due componenti fanno parte di una struttura dichiarata nel modo seguente:

```
struct point {
    int x;
    int y;
};
```

La parola chiave `struct` introduce la dichiarazione di una struttura, che è un elenco di dichiarazioni racchiuse tra parentesi graffe. Un nome, detto *contrassegno della struttura* (*structure tag*), può, a discrezione, seguire la parola `struct` (in questo caso è `point`). Il contrassegno denota l'intera struttura, e si presta dunque a fungere da abbreviazione dei dati fra parentesi.

Le variabili menzionate in una struttura sono chiamate *membri*. Un membro o un contrassegno di una struttura possono avere lo stesso nome di una variabile ordinaria (che non sia, cioè, membro di alcuna struttura) senza generare conflitti, perché è sempre possibile distinguere in base al contesto. Va anche ricordato che il nome di un membro può ripetersi in strutture diverse, sebbene sia ragionevole aspettarsi l'uso degli stessi nomi solo per oggetti con forti affinità.

Una dichiarazione `struct` definisce un tipo. La parentesi graffa chiusa al termine dell'elenco dei membri può essere seguita da una lista di variabili, come per ogni tipo fondamentale. Come dire che

```
struct { ... } x, y, z;
```

è sintatticamente analoga a

```
int x, y, z;
```

nel senso che ciascuna istruzione dichiara `x`, `y` e `z` variabili del predetto tipo e determina l'accantonamento dello spazio di memoria a esse dovuto.

La dichiarazione di una struttura non seguita da una lista di variabili non riserva invece alcuno spazio di memoria; essa si limita a descrivere uno schema o la forma di una struttura. Se, tuttavia, la dichiarazione reca un contrassegno, esso potrà poi essere impiegato per definire occorrenze della struttura.

Per esempio, data la dichiarazione di `point` precedente,

```
struct point pt;
```

definisce una variabile `pt` che è una struttura di tipo `struct point`. Una struttura può essere inizializzata facendo seguire alla sua definizione una lista di inizializzatori che devono essere espressioni costanti:

```
struct point maxpt = { 320, 200 };
```

Una struttura automatica può anche essere inizializzata mediante assegnamento o chiamando una funzione che restituisca una struttura del tipo richiesto.

Un membro di una struttura è identificato in un'espressione da un costrutto di questa forma:

*nome-struttura.membro*

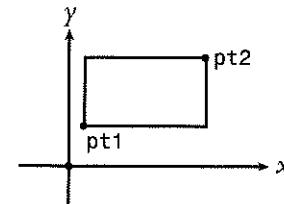
L'operatore `.` collega il nome della struttura e il nome del membro. Per visualizzare le coordinate del punto `pt`, per esempio, si avrà

```
printf("%d,%d", pt.x, pt.y);
```

e per calcolare la distanza dall'origine  $(0,0)$  a `pt`

```
double dist, sqrt(double);
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Le strutture possono essere annidate. Una rappresentazione di un rettangolo è una copia di punti che denotano gli angoli diagonalmente opposti:



```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

La struttura `rect` contiene due strutture `point`. Se dichiariamo `screen` come

```
struct rect screen;
```

allora

```
screen.pt1.x
```

si riferisce alla coordinata  $x$  del membro `pt1` di `screen`.

## 6.2 Strutture e funzioni

Le uniche operazioni legali su una struttura sono: l'esecuzione di copie o assegnamenti dell'intera struttura, la richiesta del suo indirizzo tramite & e l'accesso ai suoi membri. La copia e l'assegnamento comprendono il passaggio di argomenti alle funzioni e la restituzione di valori dalle funzioni. Le strutture non possono essere confrontate. Si può inizializzare una struttura per mezzo di un elenco di valori costanti dei membri; anche una struttura automatica può essere inizializzata, con un assegnamento.

Approfondiamo lo studio delle strutture scrivendo alcune funzioni per la manipolazione di punti e rettangoli. Gli approcci possibili sono almeno tre: passare le componenti una per volta, passare un'intera struttura o passare un puntatore a essa. Ognuno di essi ha i suoi lati positivi e negativi.

La prima funzione, `makepoint`, prende due interi e restituisce una struttura `point`:

```
/* makepoint: costruisce un punto a partire dalle componenti x e y */
struct point makepoint(int x, int y)
{
    struct point temp;

    temp.x = x;
    temp.y = y;
    return temp;
}
```

Si noti che non c'è conflitto tra il nome dell'argomento e il membro omonimo; ripetere i nomi sottolinea piuttosto la relazione tra i due.

La funzione `makepoint` può ora essere usata per inizializzare dinamicamente una struttura, o per passare strutture a una funzione:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```

Il passo successivo è un gruppo di funzioni da impiegare per i calcoli sui punti. Per esempio,

```
/* addpoint: somma due punti */
struct point addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

In questo caso sono strutture sia gli argomenti che il valore ottenuto. Anziché usare una variabile provvisoria esplicita, abbiamo incrementato le componenti di `p1` per evidenziare che le strutture sono passate per valore come tutti i parametri.

Un altro esempio è la funzione `ptinrect`, che determina se un punto si trovi all'interno di un rettangolo; stabiliamo per convenzione che un rettangolo comprenda i lati sinistro e inferiore, ma non i lati destro e superiore:

```
/* ptinrect: restituisce 1 se p appartiene a r, altrimenti 0 */
int ptinrect(struct point p, struct rect r)
{
    return p.x >= r.pt1.x && p.x < r.pt2.x
        && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

Ciò presuppone che il rettangolo sia rappresentato in una forma canonica: le coordinate di `pt1` devono essere minori delle coordinate di `pt2`. La seguente funzione restituisce un dato rettangolo in forma canonica:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

/* canonrect: porta le coordinate di un rettangolo in forma canonica */
struct rect canonrect(struct rect r)
{
    struct rect temp;

    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max(r.pt1.y, r.pt2.y);
    return temp;
}
```

Dovendo passare una struttura di grandi dimensioni a una funzione, in genere è più efficiente passare un puntatore che copiare l'intera struttura. I puntatori alle strutture sono del tutto analoghi ai puntatori alle variabili ordinarie. La dichiarazione

```
struct point *pp;
```

afferma che `pp` è un puntatore a una struttura del tipo `struct point`. Se `pp` punta a una struttura `point`, `*pp` è la struttura, mentre `(*pp).x` e `(*pp).y` ne sono i membri. Per usare `pp`, potremmo scrivere, per esempio,

```
struct point origin, *pp;

pp = &origin;
printf("L'origine e' (%d,%d)\n", (*pp).x, (*pp).y);
```

Le parentesi sono necessarie in `(*pp).x` perché l'operatore per l'accesso ai membri . ha diritto di precedenza su \*. L'espressione `*pp.x` significa `*(pp.x)`, che qui è scorretta perché `x` non è un puntatore.

I puntatori alle strutture sono così frequenti da rendere opportuna la seguente notazione abbreviata. Se *p* è un puntatore a una struttura,

*p->membro-della-struttura*

si riferisce al membro specificato. (L'operatore *->* è un segno meno seguito immediatamente da *>*.) Potremmo dunque scrivere

```
printf("L'origine e' (%d,%d)\n", pp->x, pp->y);
```

Sia *.* che *->* associano da sinistra a destra; quindi se

```
struct rect r, *rp = &r;
```

queste quattro espressioni sono equivalenti:

```
r.pt1.x  
rp->pt1.x  
(r.pt1).x  
(rp->pt1).x
```

Gli operatori *.* e *->*, insieme a *()* per le chiamate di funzione e *[]* per l'accesso agli elementi di un vettore, hanno diritto supremo di precedenza. Per esempio, data la dichiarazione

```
struct {  
    int len;  
    char *str;  
} *p;  
  
segue che  
  
++p->len
```

incrementa *len*, e non *p*, perché l'espressione equivale a *++(p->len)*. Per alterare i diritti di precedenza occorre usare le parentesi: *(++p)->len* incrementa *p* prima di accedere a *len*, e *(p++)->len* incrementa *p* dopo l'accesso; le parentesi non sono necessarie in quest'ultimo caso.

Allo stesso modo, *\*p->str* denota l'oggetto a cui punta *str*, qualunque esso sia; *\*p->str++* incrementa *str* dopo aver eseguito l'accesso all'oggetto da esso puntato (esattamente come *\*s++*); *(\*p->str)++* incrementa l'oggetto puntato da *str*; e *\*p++->str* incrementa *p* dopo aver eseguito l'accesso all'oggetto puntato da *str*.

### 6.3 Vettori di strutture

Passiamo a scrivere un programma per calcolare le occorrenze delle parole chiave del C. Servirà un vettore di stringhe di caratteri che contenga le parole chiave, e un vettore di interi per i calcoli. È possibile usare due vettori paralleli, *keyword* e *keycount*, come in

```
char *keyword[NKEYS];  
int keycount[NKEYS];
```

Proprio la circostanza che i vettori siano paralleli fa però pensare a un approccio diverso, cioè a un vettore di strutture. Una parola chiave e il numero delle sue occorrenze possono essere concepiti come una coppia:

```
char *word;  
int count;
```

e le diverse coppie sono raccolte in un vettore. La dichiarazione della struttura

```
struct key {  
    char *word;  
    int count;  
} keytab[NKEYS];
```

dichiara il tipo *struct key*, definisce un vettore *keytab* di strutture di questo tipo e alloca la memoria necessaria. Ogni elemento del vettore è una struttura. La dichiarazione può essere anche scritta così:

```
struct key {  
    char *word;  
    int count;  
};  
  
struct key keytab[NKEYS];
```

Posto che la struttura *keytab* contenga un insieme costante di nomi, la via più semplice è renderla una variabile esterna e inizializzarla una volta per tutte quando sia definita. La struttura è inizializzata in maniera analoga a quanto visto in precedenza, con una lista di inizializzatori tra parentesi graffe che segue la definizione:

```
struct key {  
    char *word;  
    int count;  
} keytab[] = {  
    "auto", 0,  
    "break", 0,  
    "case", 0,  
    "char", 0,  
    "const", 0,  
    "continue", 0,  
    "default", 0,  
    /* ... */  
    "unsigned", 0,  
    "void", 0,  
    "volatile", 0,  
    "while", 0  
};
```

Gli inizializzatori sono elencati a coppie, in modo da riflettere la disposizione dei membri della struttura.

Sarebbe più preciso applicare le parentesi graffe a ogni coppia, come in

```
{ "auto", 0 },
{ "break", 0 },
{ "case", 0 },
...
```

ma le parentesi interne non sono necessarie quando gli inizializzatori sono semplici variabili o stringhe di caratteri, sempre che l'elenco degli inizializzatori sia completo. Come di consueto, la dimensione del vettore keytab, qualora non esplicitamente indicata, sarà desunta dagli inizializzatori, se ve ne sono.

Il programma per contare le parole chiave inizia con la definizione di keytab. Per leggere i dati in ingresso, la funzione principale chiama ripetutamente una funzione getword, che tratta una parola per volta. Ciascuna parola viene cercata all'interno di keytab con una versione della funzione di ricerca binaria descritta nel Capitolo 3. Le parole chiave devono essere disposte in ordine alfabetico nella tabella.

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

#define MAXWORD 100

int getword(char *, int);
int binsearch(char *, struct key *, int);

/* conta le parole chiave C nel testo in ingresso */
main()
{
    int n;
    char word[MAXWORD];

    while (getword(word, MAXWORD) != EOF)
        if ((n = binsearch(word, keytab, NKEYS)) >= 0)
            keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n", keytab[n].count, keytab[n].word);
    return 0;
}

/* binsearch: trova la parola word in tab[0]...tab[n-1] */
int binsearch(char *word, struct key tab[], int n)
{
    int cond;
    int low, high, mid;

    low = 0;
    high = n - 1;
```

```
        while (low <= high) {
            mid = (low+high) / 2;
            if ((cond = strcmp(word, tab[mid].word)) < 0)
                high = mid - 1;
            else if (cond > 0)
                low = mid + 1;
            else
                return mid;
        }
    return -1;
}
```

La funzione getword verrà presentata tra poco; per il momento basterà dire che ogni chiamata a getword trova una parola, che è copiata nel vettore passato come suo primo argomento.

La quantità NKEYS è il numero di parole chiave contenute in keytab. Sebbene si possa ottenere questo valore manualmente, è molto più facile e sicuro farlo per mezzo del calcolatore, soprattutto se l'elenco è suscettibile di cambiamenti. Un modo per eseguire il calcolo è di mettere in coda alla lista di inizializzatori un puntatore nullo, per poi scandire tramite un ciclo keytab finché non se ne trovi la fine.

Questa procedura, però, è inutilmente complicata, visto che la dimensione del vettore è interamente determinata in fase di compilazione. La dimensione del vettore è pari al prodotto della dimensione di un elemento per il numero di elementi; quindi, il numero di elementi è semplicemente

*dimensione di keytab / dimensione di struct key*

Il C fornisce l'operatore unario `sizeof` per calcolare la dimensione di qualsiasi oggetto. Le espressioni

`sizeof oggetto`  
e

`sizeof (nome di un tipo)`

producono un intero uguale alla dimensione, espressa in byte, dell'oggetto o tipo in questione. (Per la precisione, `sizeof` produce un valore intero senza segno, il cui tipo, `size_t`, è definito nell'intestazione `<stddef.h>`.) Un oggetto può essere una variabile o un vettore o una struttura. Un nome di tipo può essere il nome di un tipo fondamentale quale `int` o `double`, o di un tipo derivato come una struttura o un puntatore.

Nel nostro caso, il numero delle parole chiave è dato dalla dimensione del vettore diviso la dimensione di un elemento. Il calcolo è eseguito da una direttiva `#define` per impostare il valore di NKEYS:

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

Un'operazione equivalente consiste nel dividere la dimensione del vettore per la dimensione di un particolare elemento.

```
#define NKEYS (sizeof keytab / sizeof(keytab[0]))
```

Questa modalità di calcolo ha il vantaggio di non dipendere dallo specifico tipo di keytab.

L'istruzione `sizeof` non può essere usata in una direttiva `#if`, perché il preprocessore non esegue l'analisi sintattica dei nomi dei tipi; l'espressione contenuta da `#define` non è però valutata dal preprocessore, e dunque il codice è legale.

E ora occupiamoci di `getword`. La stesura che presentiamo è più generale di quanto richieda questo programma, ma non complicata: `getword` legge la “parola” successiva in ingresso, dove per parola si intende una stringa di lettere e cifre che comincia con una lettera, o un carattere singolo che non sia uno spazio. Il valore della funzione è il primo carattere della parola, o `EOF` se si è raggiunta la fine del file, oppure il singolo carattere che costituisce la parola qualora non si tratti di una lettera.

```
/* getword: legge la parola o il carattere successivo in ingresso */
int getword(char *word, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = word;

    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
    for ( ; --lim > 0; w++)
        if (!isalnum(*w = getch())) {
            ungetch(*w);
            break;
        }
    *w = '\0';
    return word[0];
}
```

Vengono utilizzate le funzioni `getch` e `ungetch` scritte nel Capitolo 4. Al termine della lettura di una stringa alfanumerica, `getword` ha letto un carattere di troppo: la chiamata a `ungetch` restituisce quel carattere al flusso in ingresso in vista della chiamata successiva. La funzione `getword` utilizza anche `isspace` per ignorare gli spazi, `isalpha` per identificare le lettere, e `isalnum` per riconoscere lettere e cifre; tutte queste funzioni sono definite nell'intestazione `<ctype.h>`.

**Esercizio 6.1** La nostra versione di `getword` non è del tutto soddisfacente in merito a sottolineature, costanti stringa, commenti o direttive al preprocessore. Si scriva una versione migliore.

## 6.4 Puntatori a strutture

Per chiarire le tematiche relative ai puntatori alle strutture e ai vettori di strutture, riscriviamo il programma che conta le parole chiave, usando questa volta i puntatori anziché gli indici dei vettori.

La dichiarazione esterna di `keytab` rimane identica, contrariamente a `main` e `binsearch` che hanno bisogno di modifiche.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* conta le parole chiave C nel testo in ingresso; versione coi puntatori */
main()
{
    char word[MAXWORD];
    struct key *p;

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((p=binsearch(word, keytab, NKEYS)) != NULL)
                p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s\n", p->count, p->word);
    return 0;
}

/* binsearch: trova la parola word in tab[0]...tab[n-1] */
struct key *binsearch(char *word, struct key *tab, int n)
{
    int cond;
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;

    while (low < high) {
        mid = low + (high-low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}
```

Sono molti i particolari che meritano attenzione. Per prima cosa, la dichiarazione di `binsearch` deve indicare che la funzione restituisce un puntatore a `struct key` anziché un intero; la cosa è attestata sia dal prototipo della funzione che da `binsearch` stessa. Se `binsearch` trova la parola, restituirà un puntatore a essa, altrimenti restituirà `NULL`.

In secondo luogo, l'accesso agli elementi di `keytab` avviene ora per mezzo di puntatori, il che comporta cambiamenti notevoli al codice di `binsearch`.

Gli inizializzatori di `low` e `high` sono adesso puntatori all'inizio della tabella e alla prima locazione appena dopo di essa.

Il calcolo dell'elemento mediano non può più risultare semplicemente da

```
mid = (low+high) / 2 /* ERRATO */
```

perché l'addizione di due puntatori è illegale. La sottrazione, invece, è legale, quindi `high-low` dà il numero di elementi, e di conseguenza

```
mid = low + (high-low) / 2
```

fa sì che `mid` punti all'elemento a metà strada tra `low` e `high`.

La modifica più importante è all'algoritmo, che deve essere adattato in modo da non generare un puntatore illegale o da tentare l'accesso a un elemento inesistente nel vettore. Il problema nasce dal fatto che `&tab[-1]` e `&tab[n]` sono entrambi al di fuori dei limiti del vettore `tab`. Il primo dei due è di per sé stesso illegale; è invece illegale dereferenziare il secondo, ma la definizione del linguaggio garantisce la correttezza dell'aritmetica che coinvolga il primo elemento al di là del limite di un vettore (cioè, `&tab[n]`).

Abbiamo scritto in `main`

```
for (p = keytab; p < keytab + NKEYS; p++)
```

Se `p` è un puntatore a una struttura, le operazioni aritmetiche concernenti `p` tengono conto delle dimensioni della struttura, ragion per cui `p++` incrementa `p` di quel tanto necessario a individuare l'elemento successivo del vettore di strutture, e la condizione interrompe il ciclo al momento giusto.

Non si creda, tuttavia, che la dimensione di una struttura sia la somma delle dimensioni dei propri membri. A causa dei requisiti di allineamento per i diversi oggetti, in una struttura potrebbero esserci "buchi" anonimi imprevisti. A titolo di esempio, se la dimensione di `char` è un byte e quella di `int` quattro byte, la struttura

```
struct {
    char c;
    int i;
};
```

potrebbe ben richiedere otto byte, e non cinque. L'operatore `sizeof` restituisce il valore corretto.

Per concludere, un'ultima considerazione sulla presentazione tipografica del programma: quando una funzione restituisce un tipo complicato come un puntatore a una struttura, per esempio in

```
struct key *binsearch(char *word, struct key *tab, int n)
```

il nome della funzione è difficilmente visibile o rintracciabile con un editor di testo. Ecco perché talvolta si usa uno stile di presentazione alternativo:

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

È una questione di gusto personale; il lettore decida quale stile adottare e lo applichi uniformemente.

## 6.5 Strutture autoreferenziali

Si supponga di voler affrontare il problema più generale di contare le occorrenze di *tutte* le parole di un dato testo in ingresso. Poiché la lista delle parole non è nota in anticipo, non possiamo ordinarla al fine di applicare una ricerca binaria. E neppure possiamo fare una ricerca lineare per ogni parola letta, per stabilire se sia già stata rilevata; il programma diventerebbe troppo lento. (Più precisamente, è probabile che il suo tempo di esecuzione crescebbe quadraticamente nel numero delle parole in ingresso.) Come far fronte con efficienza a una lista di parole arbitrarie?

Una possibile soluzione è di mantenere l'insieme delle parole rilevate costantemente ordinato, collocando ogni nuova parola al posto giusto non appena la si legge. Non si può, però, realizzare l'ordinamento spostando le parole da un punto all'altro di un vettore lineare, perché anche questo metodo richiede troppo tempo. Ricorreremo invece a una struttura dei dati detta *albero binario*.

L'albero contiene un "nodo" per ogni singola parola; ciascun nodo contiene

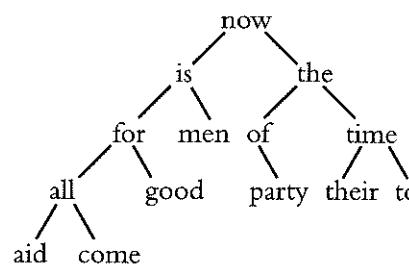
- un puntatore al testo della parola
- un conteggio del numero di occorrenze
- un puntatore al figlio sinistro, anch'esso un nodo
- un puntatore al figlio destro, anch'esso un nodo

I nodi possono avere al massimo due figli; è anche possibile che ne abbiano soltanto uno o nessuno.

I nodi sono organizzati in modo che il sottoalbero sinistro contenga solo parole lessicograficamente minori della parola che porta al nodo, e il sottoalbero destro solo parole di dimensioni maggiori. Nella pagina seguente è illustrato l'albero relativo alla frase "now is the time for all good men to come to the aid of their party"<sup>(1)</sup>, costruito inserendo in sequenza una parola per volta.

---

1. *N.d.R.* "Ora è il tempo per tutti gli uomini generosi di venire in soccorso del loro partito." Questa frase è stata per decenni un tradizionale esercizio di dattilografia negli Stati Uniti, ed è molto conosciuta nel mondo anglosassone.



Per scoprire se una parola nuova è già nell'albero, si parte dalla radice e si confronti la parola nuova con la parola presente in quel nodo. Se sono uguali, la ricerca termina con successo. Se la parola nuova è lessicograficamente minore della parola nell'albero, si continua a cercare nel figlio sinistro; se invece essa è maggiore, si prosegue la ricerca lungo quello destro. Se nella direzione richiesta non vi sono figli, la parola nuova non è nell'albero, ed è proprio il figlio mancante il posto giusto dove inserirla. Questa procedura è ricorsiva, perché la ricerca a partire da qualunque nodo sfrutta un'analogia ricerca applicata a uno dei suoi figli. Di conseguenza, per l'inserimento e la visualizzazione degli elementi saranno adatti degli algoritmi ricorsivi.

Un nodo può essere rappresentato come una struttura di quattro componenti:

```

struct tnode {      /* il nodo dell'albero: */
    char *word;        /* punta alla parola */
    int count;         /* numero delle occorrenze della parola */
    struct tnode *left; /* figlio sinistro */
    struct tnode *right; /* figlio destro */
};
  
```

Questa dichiarazione ricorsiva di un nodo potrebbe apparire rischiosa, ma è corretta. Anche se in C non sono ammesse strutture che contengano occorrenze di se stesse, la dichiarazione

```

struct tnode *left;
  
```

stabilisce che `left` è un puntatore a `tnode`, non che sia esso stesso `tnode`.

Talvolta può essere utile una variante delle strutture autoreferenziali: due diverse strutture che si riferiscono l'una all'altra. Questo è il modo di realizzarle:

```

struct t {
    ...
    struct s *p; /* p punta a un s */
};

struct s {
    ...
    struct t *q; /* q punta a un t */
};
  
```

Il codice necessario per completare il programma è sorprendentemente breve, grazie ad alcune funzioni ausiliarie, come `getword`, che abbiamo già scritto. La funzione principale legge le parole grazie a `getword` e le inserisce nell'albero tramite `addtree`.

```

#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);

int getword(char *, int);

/* computa la frequenza delle parole nel testo in ingresso */
main()
{
    struct tnode *root;
    char word[MAXWORD];

    root = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);
    return 0;
}
  
```

La funzione `addtree` è ricorsiva. La parola passata da `main` è confrontata inizialmente con il livello più alto (la radice) dell'albero; a ogni fase, quella parola è confrontata con la parola presente in quel nodo, e poi fatta discendere lungo il ramo sinistro o il ramo destro con una chiamata ricorsiva ad `addtree`. Prima o poi, la parola corrisponde a qualcosa già presente nell'albero (nel qual caso è incrementato il contatore relativo) o incontra un puntatore nullo, il che impone la creazione di un nuovo nodo dell'albero. In questa evenienza, `addtree` restituisce un puntatore al nuovo nodo, che è poi inserito nel nodo padre.

```

struct tnode *talloc(void);
char *strdup(char *);

/* addtree: aggiunge un nodo contenente w, in corrispondenza */
/*          o al di sotto del nodo p                                */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL) { /* comparsa di una nuova parola */
        p = talloc(); /* crea un nuovo nodo */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;
    else if (cond < 0)
        p->left = addtree(p->left, w);
    else
        p->right = addtree(p->right, w);
    return p;
}
  
```

```

    p->count++; /* parola ripetuta */
else if (cond < 0) /* se minore, si prende il ramo sinistro */
    p->left = addtree(p->left, w);
else /* se maggiore, si prende il ramo destro */
    p->right = addtree(p->right, w);
return p;
}

```

La memoria per il nuovo nodo è fornita dalla funzione `talloc`, che restituisce un puntatore a una zona libera delle dimensioni opportune, mentre la nuova parola è copiata da `strdup` in una locazione nascosta. Il contatore è inizializzato, e i due figli sono impostati a `NULL`. Questa sezione del codice si esegue solamente per le foglie dell'albero, nel momento in cui si aggiunge un nodo nuovo. Abbiamo omesso (poco saggiamente) il controllo degli errori nei valori prodotti da `strdup` e `talloc`.

La funzione `treeprint` visualizza il contenuto dell'albero seguendo un certo ordine: per ogni nodo, visualizza il sottoalbero sinistro (tutte le parole minori di quella corrente), poi la parola stessa, quindi il sottoalbero destro (tutte le parole più grandi). I lettori perplessi dal funzionamento della ricorsione si esercitino simulando l'esecuzione di `treeprint` sull'albero raffigurato in precedenza.

```

/* treeprint: visualizza ordinatamente il contenuto dell'albero p */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}

```

Un'indicazione pratica: se l'albero si "sbilancia" perché le parole del testo non hanno una distribuzione sufficientemente casuale, il tempo di esecuzione del programma può allungarsi eccessivamente. Nella peggiore delle ipotesi, se le parole sono già ordinate, il programma effettua una costosa simulazione della ricerca lineare. Esistono generalizzazioni di tali alberi binari che non incorrono in questo inconveniente, ma non le tratteremo in questa sede.

Prima di abbandonare questo esempio, è utile fare una breve digressione su un problema che riguarda gli allocator di memoria. Chiaramente, è auspicabile che ci sia un solo allocator di memoria all'interno di un programma, in grado di assegnare memoria a oggetti di natura diversa. Ma se uno stesso allocator deve occuparsi delle richieste, poniamo, di puntatori a caratteri e puntatori a oggetti di tipo `struct tnodes`, sorgono due ostacoli. Primo, come soddisfare il requisito di molte macchine per cui oggetti di certi tipi devono ottemperare a restrizioni di allineamento (per esempio, gli interi spesso devono avere indirizzi pari)? Secondo, quali dichiarazioni possono ovviare al fatto che un allocator deve necessariamente restituire tipi differenti di puntatori?

I requisiti di allineamento possono essere facilmente soddisfatti, in linea di massima, a prezzo di un certo spreco di spazio, controllando che l'allocator restituisca sempre un puntatore che rispetti *tutte* le restrizioni di allineamento. Visto che il programma `alloc` del Ca-

pitolo 5 non garantisce alcun allineamento particolare, impiegheremo la funzione della libreria standard `malloc`, adatta allo scopo. Nel Capitolo 8 mostreremo una possibile implementazione di `malloc`.

La questione della dichiarazione del tipo per funzioni come `malloc` è piuttosto fastidiosa per i linguaggi che trattano seriamente la verifica dei tipi. In C, il metodo corretto consiste nel dichiarare che `malloc` restituisce un puntatore a `void`, per poi forzare con un cast esplicitamente il puntatore nel tipo desiderato.<sup>(2)</sup> Sia `malloc` che le funzioni collegate sono dichiarate nell'intestazione standard `<stdlib.h>`. Dunque `talloc` può essere scritta come

```

#include <stdlib.h>

/* talloc: crea un tnode */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}

```

La funzione `strdup` si limita a copiare in una porzione sicura di memoria, ottenuta chiamando `malloc`, la stringa che riceve come argomento:

```

char *strdup(char *s) /* crea un duplicato di s */
{
    char *p;

    p = (char *) malloc(strlen(s)+1); /* +1 per '\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
}

```

La funzione `malloc` restituisce `NULL` se non c'è spazio disponibile; in questo caso, `strdup` restituisce quel valore, lasciando la gestione degli errori al chiamante.

La memoria ottenuta tramite `malloc` può essere liberata e riutilizzata con una chiamata a `free`; si vedano i Capitoli 7 e 8.

**Esercizio 6.2** Si scriva un programma che, dopo aver letto un programma in C, visualizzi in ordine alfabetico ciascun gruppo di nomi di variabili che siano identici nei primi 6 caratteri, ma diversi in qualche punto successivo. Le parole all'interno di stringhe e commenti non devono essere considerate. Si renda 6 un parametro controllabile dalla riga di comando.

2. N.d.R. Nell'errata corrigé al testo, pubblicata elettronicamente, si legge: "[Questo commento] deve essere rivisto. L'esempio [cioè, il programma appena visto] è corretto, e funziona, ma il consiglio [relativo alla conversione forzata del tipo restituito] è discutibile, alla luce degli standard ANSI/ISO del 1988-1989. La conversione forzata non è [più] necessaria [...], e potenzialmente dannosa se `malloc`, o una funzione che ne faccia le veci, non è dichiarata del tipo `void *`. Il cast esplicito può impedire il rilevamento di un errore nel codice. D'altro canto, prima dell'avvento dello standard ANSI, la conversione forzata del tipo era realmente necessaria, e continua a esserlo anche nel C++.]"

**Esercizio 6.3** Si scriva un programma per i rimandi incrociati che visualizzi un elenco di tutte le parole di un documento e, per ciascuna di esse, un elenco dei numeri delle righe in cui appare. Si escludano articoli e congiunzioni come “il”, “e” e così via.

**Esercizio 6.4** Si scriva un programma che visualizzi in ordine decrescente di frequenza le diverse parole del testo in ingresso. Ogni parola sia fatta precedere dal numero di occorrenze.

## 6.6 Ricerca su tabelle

Al fine di illustrare ulteriori aspetti delle strutture presentiamo qui il nucleo di un'applicazione per la ricerca su tabelle. Questo codice evoca ciò che tipicamente si incontra nelle funzioni di gestione della tabella dei simboli di un elaboratore di macro (come il preprocessore del C) o di un compilatore. Si prenda per esempio l'istruzione `#define`. Quando una riga come

```
#define IN 1
```

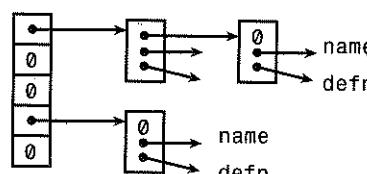
fa la sua comparsa, il nome `IN` e il testo sostitutivo `1` vengono memorizzati in una tabella. Successivamente, quando il nome `IN` figura in un'istruzione come

```
state = IN;
```

deve essere sostituito da `1`.

Vi sono due funzioni che agiscono sui nomi e sui testi sostitutivi: `install(s, t)` registra il nome `s` e il testo sostitutivo `t` in una tabella, dove `s` e `t` sono semplici stringhe di caratteri; `lookup(s)` cerca la stringa `s` nella tabella, e restituisce un puntatore alla posizione della stringa o `NULL` se essa non è presente.

L'algoritmo di ricerca è basato su una tabella hash. Il termine “hash” si riferisce al fatto che l'algoritmo associa al nome da trovare un piccolo intero non negativo, secondo una funzione che ne esegue il calcolo tramite una sorta di rimescolamento, come vedremo in dettaglio fra poco. L'intero così ottenuto, detto a volte chiave hash, è poi usato come indice per accedere agli elementi di un vettore di puntatori, i quali puntano a loro volta a una lista di blocchi corrispondenti a nomi aventi la data chiave hash; un puntatore del vettore è `NULL` se non vi sono nomi corrispondenti a quella chiave.



Un blocco è una struttura costituita da tre puntatori che indicano il nome, il relativo testo sostitutivo e il blocco successivo della lista. Quest'ultimo puntatore è nullo solo se si tratta dell'ultimo blocco della lista.

```

struct nlist { /* elemento della tabella: */
    struct nlist *next; /* elemento successivo della lista */
    char *name; /* nome definito */
    char *defn; /* testo sostitutivo */
};

};


```

Il vettore di puntatori è dato semplicemente da

```
#define HASHSIZE 101
static struct nlist *hashtab[HASHSIZE]; /* tabella dei puntatori */


```

La funzione per il calcolo della chiave hash, usata sia da `install` che da `install`, genera la chiave per passi, uno per ogni carattere del nome; ad ogni passo, aggiunge il valore numerico del carattere corrente a una combinazione rimestata (“hashed”, appunto) dei valori dei caratteri precedenti; essa restituisce poi l'intero ottenuto, modulo la dimensione del vettore. Anche se non è il migliore tra i metodi per computare una chiave hash, è conciso ed efficace.

```

/* hash: computa la chiave hash della stringa s */
unsigned hash(char *s)
{
    unsigned hashval;

    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}

```

L'uso dell'aritmetica senza segno garantisce che la chiave sia non negativa.

La chiave hash indica solo un punto di partenza per la ricerca della parola nel vettore `hashtab`: ma se la stringa cercata risiede effettivamente nella tabella, non può trovarsi che nella lista di blocchi che comincia da quel punto. La ricerca completa è eseguita da `lookup`, che restituisce un puntatore alla posizione della stringa o il valore `NULL` se la stringa non è presente.

```

/* lookup: cerca s nella tabella hash */
struct nlist *lookup(char *s)
{
    struct nlist *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np; /* trovata */
    return NULL; /* non trovata */
}

```

Il ciclo `for` in questa funzione è la tecnica classica per la scansione di una lista:

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
    ...

```

La funzione `install` sfrutta `lookup` per stabilire se il nome da inserire nell'albero sia già presente. In caso affermativo, la nuova definizione sostituirà la vecchia; altrimenti, si costruisce un nuovo nodo. La funzione restituisce `NULL` se, per una qualunque ragione, non vi è spazio sufficiente per il nuovo nodo.

```
struct nlist *lookup(char *);
char * strdup(char *);

/* install: inserisce (name, defn) nella tabella hash */
struct nlist *install(char *name, char *defn)
{
    struct nlist *np;
    unsigned hashval;

    if ((np = lookup(name)) == NULL) { /* non trovato */
        np = (struct nlist *) malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
        hashval = hash(name);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    } else      /* il nome e' gia' presente */
        free((void *) np->defn); /* dealloca la defn precedente */
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}
```

**Esercizio 6.5** Si scriva la funzione `undef`, che rimuova un nome e la sua definizione dalla tabella gestita da `lookup` e `install`.

**Esercizio 6.6** Sulla base delle funzioni di questo paragrafo, si realizzi un semplice elaboratore delle direttive `#define` prive di argomenti adatto per i programmi in C. Potranno tornare utili le funzioni `getch` e `ungetch`.

## 6.7 Typedef

Il C fornisce la funzionalità `typedef` per assegnare nuovi nomi ai tipi dei dati. Per esempio, la dichiarazione

```
typedef int Length;
```

fa del nome `Length` un sinonimo di `int`. Il tipo `Length` si usa in dichiarazioni, conversioni di tipo e così via, esattamente con le stesse modalità del tipo `int`:

```
Length len, maxlen;
Length *lengths[];
```

In maniera analoga, la dichiarazione

```
typedef char *String;
```

rende `String` sinonimo di `char *`, cioè di un puntatore a carattere, che può quindi essere usato in dichiarazioni e conversioni:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Si osservi che il tipo dichiarato da `typedef` assume la posizione di un nome di variabile, e non compare subito dopo la parola `typedef`. Sul piano sintattico, `typedef` è analogo agli attributi `extern`, `static`, eccetera. Abbiamo usato nomi in maiuscolo per far risaltare le definizioni.

Per un esempio più complesso, potremmo applicare `typedef` ai nodi dell'albero presentato nel Paragrafo 6.5:

```
typedef struct tnode *Treenode;
typedef struct tnode { /* il nodo dell'albero: */
    char *word;           /* punta alla parola */
    int count;            /* numero delle occorrenze della parola */
    Treenode left;        /* figlio sinistro */
    Treenode right;       /* figlio destro */
} Treenode;
```

Ciò genera due nuove parole chiave, `Treenode` (una struttura) e `Treenode` (un puntatore a una struttura); entrambe le parole rappresentano tipi. Quindi la routine `talloc` potrebbe diventare

```
Treenode talloc(void)
{
    return (Treenode) malloc(sizeof(Treenode));
}
```

È bene sottolineare che una dichiarazione `typedef` non produce un nuovo tipo in nessun senso, ma definisce solo un nuovo nome per qualche tipo esistente. Né ci sono novità semantiche: le variabili dichiarate in questo modo abbreviato possiedono esattamente le stesse proprietà delle variabili le cui dichiarazioni siano scritte per esteso. In effetti, `typedef` è analoga a `#define`, salvo il fatto che, essendo interpretata dal compilatore, sono alla sua portata sostituzioni che il preprocessore non è in grado di gestire. Così, l'istruzione

```
typedef int (*PFI)(char *, char *);
```

genera il tipo `PFI`, che sta per “puntatore alla funzione (di due argomenti `char *`) che restituisce `int`”. Nel programma per l'ordinamento presentato nel Capitolo 5, per esempio, si sarebbe potuta applicare questa definizione in contesti quali

```
PFI strcmp, numcmp;
```

Prescindendo da questioni puramente estetiche, vi sono due ragioni fondamentali per l'utilizzo di `typedef`. La prima è rendere un programma parametrico per risolvere questioni di portabilità. Dare nuovi nomi simbolici ai tipi di dati dipendenti dalla macchina permette di adattare il programma a macchine diverse tramite la modifica delle sole istruzioni `type-`

`def`. Secondo una prassi comune, si applica `typedef` a varie quantità intere, quindi si sceglie opportunamente tra i tipi `short`, `int` e `long` per ciascuna macchina: sono possibili esempi tipi quali `size_t` e `ptrdiff_t` della libreria standard.

La seconda ragione che legittima l'uso di `typedef` è una maggiore chiarezza del codice: un tipo denominato `Treeptr` può risultare più comprensibile di un puntatore a una struttura complicata.

## 6.8 Unioni

Si definisce *unione* una variabile che può contenere (in momenti diversi) oggetti di tipo e dimensioni differenti, con il compilatore che si fa carico di tenere traccia di dimensioni e allineamenti. Le unioni costituiscono un modo di intervenire su dati di diverso genere all'interno di un'unica area di memoria, senza includere nel programma alcuna informazione vincolata alla macchina. Si tratta di un costrutto analogo ai record varianti del Pascal.

Con un esempio che ricorre comunemente nella gestione della tabella dei simboli da parte di un compilatore, ipotizziamo che una costante possa essere di tipo `int`, `float`, o un puntatore a carattere. Il valore di una costante specifica deve essere memorizzato in una variabile del tipo adeguato, sebbene la gestione della tabella sia più conveniente se ogni valore occupa la stessa quantità di memoria e se la sua collocazione non dipende dal tipo. È appunto questa la finalità di un'unione: una singola variabile che può contenere legittimamente un'altra variabile di diverso tipo. La sintassi si fonda sulle strutture:

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

La variabile `u` sarà abbastanza grande da contenere il più grande dei tre tipi; la dimensione precisa dipende dall'implementazione. Ognuno di questi tre tipi può essere assegnato a `u` e poi impiegato nelle espressioni, purché lo si faccia coerentemente: il tipo usato deve essere quello memorizzato per ultimo. È responsabilità del programmatore tenere traccia di quale tipo sia di volta in volta memorizzato in un'unione; se si memorizza un certo tipo per poi usarlo come un altro, gli effetti dipenderanno dall'implementazione.

Sotto il profilo sintattico, l'accesso ai membri di un'unione è dato da

`nome-unione.membro`

oppure da

`puntatore-unione->membro`

come per le strutture. Se la variabile `utype` tiene traccia del tipo attualmente memorizzato in `u`, sarà facile imbattersi in codice quale:

```
if (utype == INT)
    printf("%d\n", u.ival);
```

```
else if (utype == FLOAT)
    printf("%f\n", u.fval);
else if (utype == STRING)
    printf("%s\n", u.sval);
else
    printf("tipo %d non ammesso in utype\n", utype);
```

Si possono trovare unioni all'interno di strutture e vettori, e viceversa. La notazione per accedere a un membro di un'unione in una struttura (o viceversa) è identica a quella per le strutture annidate. Per esempio, nel vettore di strutture definito da

```
struct {
    char *name;
    int flags;
    int utype;
    union {
        int ival;
        float fval;
        char *sval;
    } u;
} symtab[NSYM];
```

ci si riferisce al membro `ival` come

`symtab[i].u.ival`

e al primo carattere della stringa `sval` tramite una delle due forme

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

In effetti, un'unione è una struttura in cui tutti i membri hanno scostamento nullo dalla base, la struttura è sufficientemente grande per contenere il membro "più ampio", e l'allineamento è appropriato per tutti i tipi dell'unione. Le operazioni permesse sulle unioni e le strutture sono le stesse: l'esecuzione di copie o assegnamenti dell'unione, la richiesta di indirizzo tramite `&` e l'accesso a un membro.

Un'unione può essere inizializzata solo con un valore del tipo del proprio membro iniziale; perciò l'unione `u` può essere inizializzata esclusivamente con un valore intero.

Il programma di allocazione illustrato nel Capitolo 8 mostrerà come servirsi di un'unione per forzare una variabile ad allinearsi a particolari vincoli di memoria.

## 6.9 Campi di bit

Quando lo spazio di memoria è un bene prezioso, può rendersi necessario collocare molti oggetti all'interno di una singola parola della macchina. Un caso classico è l'uso di un insieme di flag di un solo bit in applicazioni quali la tabella dei simboli di un compilatore. Anche il trattamento di dati il cui formato è imposto dall'esterno, per esempio da un'interfaccia a un dispositivo periferico, richiede spesso la capacità di raggiungere singole parti di una parola.

Si immagini un frammento di un compilatore che manipola una tabella dei simboli. Gli identificatori in un programma hanno determinati attributi: possono essere parola chiave o meno, identificatori esterni e/o statici, e via dicendo. Il modo più sintetico di codificare queste informazioni è un insieme di flag, raccolto in un unico carattere o intero.

Di solito si inizia con la definizione di un insieme di maschere che corrispondono alle posizioni dei bit rilevanti, come in

```
#define KEYWORD 01
#define EXTERNAL 02
#define STATIC 04

enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

I numeri devono essere potenze di due. A questo punto accedere ai bit diventa una questione di applicazione degli operatori, descritti nel Capitolo 2, che servono a farli scorrere, a mascherarli e a prenderne il complemento.

Alcune espressioni appaiono di frequente:

```
flags |= EXTERNAL | STATIC;
```

attiva i bit EXTERNAL e STATIC all'interno di flags, laddove

```
flags &= ~(EXTERNAL | STATIC);
```

li pone a zero, e

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

è vera se entrambi i bit sono zero.

Il C offre in alternativa la possibilità di definire e accedere a campi all'interno di una parola per via diretta anziché mediata da operatori logici bit per bit. Un *campo di bit*, o semplicemente *campo*, è un insieme di bit contigui contenuti in un'unità di memoria definita dall'implementazione che chiameremo convenzionalmente "parola". La sintassi che definisce il campo e ne consente l'accesso si basa sulle strutture. Per esempio, le direttive #define analizzate precedentemente potrebbero essere sostituite con la definizione di tre campi:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern : 1;
    unsigned int is_static : 1;
} flags;
```

Questo definisce una variabile di nome flags contenente tre campi da un bit. Il numero che fa seguito ai due punti rappresenta la dimensione del campo in bit. I campi sono dichiarati unsigned int per assicurare che siano quantità prive di segno.

Ai campi singoli si accede come agli altri membri di una struttura: flags.is\_keyword, flags.is\_extern e così via. I campi si comportano come piccoli interi e possono far parte delle espressioni aritmetiche così come gli altri interi.

Ne consegue che i precedenti esempi si possono formulare più naturalmente come:

```
flags.is_extern = flags.is_static = 1;
```

per attivare i bit;

```
flags.is_extern = flags.is_static = 0;
```

per disattivarli; e

```
if (flags.is_extern == 0 && flags.is_static == 0)
    ...
```

per verificarne lo stato.

Quasi tutto ciò che concerne i campi dipende dall'implementazione. Se un campo possa sovrapporsi alla fine di una parola è definito dall'implementazione. Non vi è necessità di un nome per i campi; i campi anonimi (solo due punti e la dimensione) sono usati come riempitivo. La dimensione speciale 0 può essere usata per forzare l'allineamento alla parola successiva.

Su alcune macchine i campi sono assegnati da sinistra a destra, e su altre al contrario. Ciò significa che, sebbene i campi siano utili per trattare dati il cui formato è definito dal sistema, il problema di quale estremo venga prima deve essere ponderato con cura se i dati sono definiti dall'esterno; i programmi che dipendono da simili incognite non sono portabili. I campi possono essere dichiarati soltanto come int: per migliorare la portabilità è opportuno specificare esplicitamente signed o unsigned. Essi non sono vettori, e non hanno indirizzi, per cui non è possibile applicare loro l'operatore &.

# Input e output

LE FUNZIONALITÀ LEGATE ALL'INPUT E ALL'OUTPUT non sono parte integrante del linguaggio C, e dunque non sono state evidenziate fino a questo momento. Ciononostante, i programmi interagiscono con il loro ambiente secondo modalità molto più complesse di quelle mostrate fin qui. Questo capitolo è dedicato alla descrizione della libreria standard, un insieme di funzioni a uso dei programmi in C che vanno dal trattamento dell'input e dell'output alla manipolazione delle stringhe, dalla gestione della memoria alle funzioni matematiche, e così via. Concentreremo la nostra attenzione su input e output.

Lo standard ANSI definisce con precisione queste funzioni della libreria, con l'intento di renderle compatibili con ogni sistema che preveda il C. I programmi che limitano le proprie interazioni di sistema a funzionalità fornite dalla libreria standard possono essere trasferiti da un sistema all'altro senza subire alcuna modifica.

Le proprietà delle funzioni della libreria sono specificate in più di una dozzina di intestazioni; ne abbiamo già viste diverse, tra cui `<stdio.h>`, `<string.h>` e `<ctype.h>`. Non illustreremo qui tutta la libreria, perché ci interessa maggiormente scrivere programmi in C che ne facciano uso. La libreria è descritta dettagliatamente nell'Appendice B.

## 7.1 Standard input e output

Come scritto nel Capitolo 1, la libreria implementa un semplice modello di ingresso e uscita di dati testuali. Un flusso di testo è composto da una sequenza di righe, e ciascuna di

esse termina con un carattere newline. Se il sistema ambiente non opera in questo modo, la libreria fa tutto il possibile per simularlo. Per esempio, la libreria potrebbe convertire i caratteri di ritorno del carrello e di fine riga (“linefeed”) in carattere newline in ingresso, e ri-convertirli poi in uscita.

Il meccanismo di input più semplice è di leggere con `getchar` un carattere per volta dal cosiddetto *standard input*, che di norma è la tastiera:

```
int getchar(void)
```

La funzione `getchar` restituisce a ogni invocazione il carattere in ingresso successivo, o `EOF` nel caso incontri la fine del file. Il simbolo di costante `EOF`, definito in `<stdio.h>`, vale solitamente `-1`, ma le condizioni dovrebbero prescindere dal suo valore specifico e citare solo la forma simbolica `EOF`.

In molti ambienti un file può sostituire la tastiera grazie a un meccanismo noto come redirezione dell’input. Se il programma `prog` impiega `getchar`, il comando

```
prog <infile
```

fa sì che `prog` legga i caratteri in ingresso dal file `infile`, e non dallo standard input. La redirezione avviene in maniera tale che `prog` medesimo sia ignaro del cambiamento; in particolare, la stringa “`<infile`” non rientra fra gli argomenti della riga di comando in `argv`. La redirezione del flusso in ingresso resta invisibile anche se i nuovi dati provengono da un altro programma attraverso un meccanismo di concatenazione fra l’output e l’input noto come “pipe”: su alcuni sistemi, la riga di comando

```
otherprog | prog
```

esegue i due programmi `otherprog` e `prog`, e incanala lo standard output di `otherprog` nello standard input di `prog`.

La funzione

```
int putchar(int)
```

è usata per generare dati in uscita: `putchar(c)` immette il carattere `c` nello *standard output*, che corrisponde di default allo schermo. La funzione restituisce il carattere visualizzato, o `EOF` in caso di errore. Tuttavia, l’output può solitamente essere deviato verso un file mediante `>nomefile`: se `prog` usa `putchar`,

```
prog >outfile
```

lo standard output di `prog` diventa `outfile`. Se il meccanismo pipe è disponibile,

```
prog | anotherprog
```

incanala lo standard output di `prog` nello standard input di `anotherprog`.

I dati in uscita prodotti da `printf` trovano anch’essi il modo di raggiungere lo standard output. Le chiamate a `putchar` e a `printf` possono essere alternate; i dati in uscita seguiranno l’ordine delle chiamate.

Ogni file sorgente che contenga riferimenti a funzioni della libreria preposte all’input o all’output deve includere la riga

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

in un punto che preceda il primo riferimento. Quando il nome appare tra `< e >` l’intestazione è ricercata in una serie di luoghi predefiniti (per esempio, se si tratta di sistemi UNIX, solitamente nella directory `/usr/include`).

Molti programmi leggono solo un flusso in ingresso e scrivono solo un flusso in uscita; per tali programmi, `getchar`, `putchar` e `printf` si rivelano del tutto adeguati alla gestione di input e output, e di certo sufficienti per cominciare. Questo è particolarmente vero se si usa oculatamente il meccanismo della redirezione. Come esempio, si consideri il programma `lower`, che converte il proprio input in caratteri minuscoli:

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

main() /* lower: converte i dati in ingresso in minuscole */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

La funzione `tolower` è definita in `<ctype.h>`, e converte una lettera maiuscola in minuscola, lasciando invariati gli altri caratteri. Come accennato in precedenza, “funzioni” quali `getchar` e `putchar` in `<stdio.h>` e `tolower` in `<ctype.h>` sono spesso macroistruzioni, che hanno il pregio di evitare una chiamata di funzione per ogni carattere. Spiegheremo come ciò avviene nel Paragrafo 8.5. I programmi che usano le funzioni di `<ctype.h>`, indipendentemente da come esse sono implementate, non devono preoccuparsi del set di caratteri della macchina ospitante.

**Esercizio 7.1** Si scriva un programma per convertire le lettere maiuscole in minuscole, o viceversa, a seconda del nome contenuto in `argv[0]` con cui esso è invocato.

## 7.2 Formattazione dei dati in uscita: `printf`

La funzione di output `printf` traduce i valori interni in caratteri. Abbiamo usato `printf` in maniera informale nei capitoli precedenti. La presente descrizione riguarda gli usi più tipici, ma per una trattazione completa rimandiamo il lettore all’Appendice B. La funzione

```
int printf(char *format, arg1, arg2, ...);
```

converte, formatta e visualizza i propri argomenti sullo standard output, sotto il controllo di `format`. Essa restituisce il numero di caratteri visualizzati.

La stringa `format` contiene due tipi di oggetti: caratteri ordinari, che vengono copiati sul flusso in uscita, e specifiche di conversione, ciascuna delle quali determina la conversione e la visualizzazione del corrispondente argomento. Ogni specifica di conversione inizia con un `%` e termina con un carattere conversione; tra `%` e il carattere di conversione possono esserci, nell'ordine:

- Un segno meno, che segnala l'allineamento a sinistra dell'argomento convertito.
- Un numero che specifica l'ampiezza minima del campo. L'argomento convertito sarà visualizzato in un campo di ampiezza almeno pari a questa. Se necessario, saranno aggiunti degli spazi alla sua sinistra (o a destra, se si vuole l'allineamento a sinistra) per rispettare l'ampiezza del campo.
- Un punto, che separa l'ampiezza del campo dalla specifica del grado di precisione.
- Un numero, il grado di precisione, che specifica: per le stringhe, il numero massimo di caratteri visualizzabili; per i numeri con virgola mobile, il numero di cifre dopo il punto decimale; per gli interi, il numero minimo di cifre.
- `h` se l'intero deve essere visualizzato come `short`, o `l` (lettera elle) per `long`.

I caratteri di conversione sono elencati nella Tabella 7.1. Se il carattere che segue `%` non dà luogo a una specifica di conversione, il comportamento del programma è indefinito.

L'ampiezza o il grado di precisione possono essere specificati come `*`, e in questo caso il valore si calcola convertendo l'argomento successivo (che deve essere di tipo `int`). Per esempio, per visualizzare un massimo di `max` caratteri di una stringa `s`, si può scrivere

```
printf("%.*s", max, s);
```

La maggior parte delle conversioni è stata trattata nei capitoli precedenti. Un'eccezione è la specifica del grado di precisione in rapporto alle stringhe. L'elenco seguente mostra l'effetto di una serie di specifiche nella visualizzazione di "ciao mondo" (10 caratteri). Abbiamo messo i due punti per delimitare ogni campo ed evidenziarne l'estensione.

<code>:%s:</code>	<code>:ciao mondo:</code>
<code>:%8s:</code>	<code>:ciao mondo:</code>
<code>:%.8s:</code>	<code>:ciao mon:</code>
<code>:%-8s:</code>	<code>:ciao mondo:</code>
<code>:%.15s:</code>	<code>:ciao mondo:</code>
<code>:%-15s:</code>	<code>:ciao mondo :</code>
<code>:%15.8s:</code>	<code>: ciao mon:</code>
<code>:%-15.8s:</code>	<code>:ciao mon :</code>

Attenzione: `printf` usa il suo primo argomento per stabilire quanti altri argomenti aspettarsi e di quale tipo. Se non vi sono argomenti a sufficienza o se sono del tipo sbagliato farà confusione, e il lettore avrà risposte scorrette. Bisogna anche saper distinguere tra queste due chiamate:

```
printf(s);      /* FALLISCE se s contiene % */
printf("%s", s); /* VA BENE */
```

Tabella 7.1 Conversioni principali di `printf`.

CARATTERE	TIPO DELL'ARGOMENTO; STAMPATO COME
<code>d, i</code>	<code>int</code> ; numero in notazione decimale.
<code>o</code>	<code>unsigned int</code> ; numero in notazione ottale privo di segno (senza lo zero in testa).
<code>x, X</code>	<code>unsigned int</code> ; numero in notazione esadecimale (senza <code>0x</code> o <code>0X</code> in testa) privo di segno, dove <code>abcdef</code> o <code>ABCDEF</code> stanno per 10, ..., 15.
<code>u</code>	<code>unsigned int</code> ; numero in notazione decimale privo di segno.
<code>c</code>	<code>int</code> ; singolo carattere.
<code>s</code>	<code>char *</code> ; stampa i caratteri della stringa fino a raggiungere <code>\0</code> , o a esaurire il grado di precisione specificato.
<code>f</code>	<code>double</code> ; <code>[-]i.#####</code> , dove <code>i</code> è la parte intera, <code>#####</code> sono le cifre dopo il punto decimale, in numero dato dal grado di precisione specificato (di default 6).
<code>e, E</code>	<code>double</code> ; <code>[-]i.#####e±xx</code> o <code>[-]i.#####E±xx</code> , dove <code>xx</code> è l'esponente, e il numero di <code>d</code> è dato dal grado di precisione specificato (di default 6).
<code>g, G</code>	<code>double</code> ; usa <code>%e</code> o <code>%E</code> se l'esponente è minore di -4 o maggiore o uguale al grado di precisione; altrimenti usa <code>%f</code> . Gli zeri in coda, o il punto decimale in coda, non sono stampati.
<code>p</code>	<code>void *</code> ; puntatore (rappresentazione dipendente dall'implementazione).
<code>%</code>	non ha luogo alcuna conversione; visualizza un <code>%</code> .

La funzione `sprintf` compie le stesse conversioni di `printf`, ma memorizza i dati in uscita in una stringa:

```
int sprintf(char *string, char *format, arg1, arg2, ...);
```

`sprintf` compone gli argomenti `arg1`, `arg2`, ecc., come prima, secondo i criteri prescritti da `format`, ma colloca il risultato in `string` anziché inoltrarlo allo standard output; `string` deve essere di grandezza idonea ad accogliere il risultato.

**Esercizio 7.2** Si scriva un programma che visualizzi dati in ingresso arbitrari in modo ragionevole. Il programma dovrebbe quantomeno rappresentare, in notazione ottale o esadecimale a seconda della prassi del sistema ambiente, caratteri non direttamente visualizzabili, e spezzare le righe di testo troppo lunghe.

### 7.3 Liste di argomenti di lunghezza variabile

Questo paragrafo contiene un'implementazione di una versione minima di `printf`, per illustrare come si scrive una funzione che elabori una lista di argomenti di lunghezza variabile in maniera da garantire portabilità. Poiché ci interessa soprattutto la gestione degli argomenti, `minprintf` elaborerà la stringa di composizione e i suoi argomenti, ma chiamerà la vera `printf` per eseguire le conversioni.

La dichiarazione corretta di `printf` è

```
int printf(char *fmt, ...)
```

dove la ellissi `...` indica che numero e tipo degli argomenti omessi possono variare. La dichiarazione `...` può apparire soltanto alla fine di una lista di argomenti. La nostra `minprintf` è dichiarata come

```
void minprintf(char *fmt, ...)
```

dato che non restituiremo il numero di caratteri visualizzati, come invece fa `printf`.

Il difficile è capire come `minprintf` possa procedere con la lista degli argomenti quando la lista non ha neanche un nome. L'intestazione standard `<stdarg.h>` contiene una serie di macro per ovviare a questo problema. L'implementazione di questa intestazione cambierà da macchina a macchina, ma l'interfaccia che propone è uniforme.

Il tipo `va_list` serve a dichiarare una variabile che, di volta in volta, si riferisce a ciascun argomento; in `minprintf`, questa variabile è chiamata `ap`, che è l'abbreviazione di "argument pointer", ovvero "puntatore ad argomento". La macro `va_start` inizializza `ap` facendo sì che questa punti al primo argomento anonimo. La macro deve essere chiamata una volta prima di usare `ap`. È necessario che vi sia almeno un argomento con nome; l'ultimo di tali argomenti serve a `va_start` come punto di riferimento per individuare gli argomenti anonimi.

Ogni invocazione di `va_arg` restituisce un argomento e sposta `ap` al successivo; `va_arg` impiega un nome di tipo per decidere quale tipo restituire e di quanto far avanzare `ap`. Infine, `va_end` compie le operazioni di pulizia necessarie, e deve obbligatoriamente essere chiamata prima che la funzione termini.

Queste proprietà costituiscono la base della nostra `printf` in versione semplificata:

```
#include <stdarg.h>

/* minprintf: printf minimale con lista di argomenti */
/*           di lunghezza variabile                   */
void minprintf(char *fmt, ...)
{
    va_list ap; /* punta in sequenza a ogni argomento anonimo */
    char *p, *sval;
    int ival;
    double dval;

    va_start(ap, fmt); /* fa puntare ap al primo argomento anonimo */
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 'd':
            ival = va_arg(ap, int);
```

```
            printf("%d", ival);
            break;
        case 'f':
            dval = va_arg(ap, double);
            printf("%f", dval);
            break;
        case 's':
            for (sval = va_arg(ap, char *); *sval; sval++)
                putchar(*sval);
            break;
        default:
            putchar(*p);
            break;
    }
}
va_end(ap); /* fa pulizia alla fine della funzione */
```

**Esercizio 7.3** Si adatti `minprintf` in maniera che possa gestire qualche altra funzionalità di `printf`.

## 7.4 Formattazione dei dati in ingresso: `scanf`

La funzione `scanf` è analoga a `printf` in materia di input, e offre molte delle sue funzionalità di conversione in direzione opposta.

```
int scanf(char *format, ...)
```

Essa legge i caratteri in ingresso dallo standard input, li interpreta secondo le specifiche di composizione `format`, e ne memorizza i risultati negli argomenti che restano. L'argomento `format` è descritto in seguito; gli altri argomenti, *ognuno dei quali deve essere un puntatore*, indicano dove il relativo dato in ingresso convertito debba essere memorizzato. Come nel caso di `printf`, in questo paragrafo sono riassunte le caratteristiche più utili, senza pretese di completezza.

La funzione `scanf` si ferma quando esaurisce la propria stringa di composizione, o quando alcuni dati in ingresso non soddisfano le specifiche di conversione. Il valore che restituisce è costituito dal numero di oggetti in ingresso effettivamente memorizzati, il che torna utile per decidere il numero dei dati trovati. Alla fine del testo è restituito `EOF`; si noti che è diverso da zero: il valore zero significa invece che il successivo carattere in ingresso non è conforme alla prima specifica della stringa di composizione. La prossima chiamata a `scanf` riprenderà la ricerca dal primo carattere successivo a quello convertito per ultimo.

Esiste anche una funzione `sscanf` che legge da una stringa anziché dallo standard input:

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

Essa esamina la stringa in base alla specifica `format`, e memorizza i valori che ne derivano in `arg1, arg2, ecc.` Questi argomenti devono essere puntatori.

La stringa di composizione `format` generalmente comprende le specifiche di conversione, che dirigono l'analisi dei dati in ingresso. La stringa di composizione può contenere:

- Spazi bianchi o caratteri di tabulazione, che sono ignorati.
- Caratteri normali (ma non %), che si suppone corrispondano al successivo carattere diverso da uno spazio nel flusso in ingresso.
- Specifiche di conversione, costituite dal carattere %, da un carattere facoltativo \* per la soppressione dell'assegnamento, da un numero facoltativo che precisa l'ampiezza massima del campo, eventualmente da una h, l o L che indicano la dimensione dello spazio riservato alla memorizzazione del dato in ingresso, e infine da un carattere di conversione.

Una specifica di conversione presiede alla conversione del successivo campo in ingresso. Normalmente il risultato è collocato nella variabile a cui punta l'argomento corrispondente. Se la soppressione dell'assegnamento è segnalata dal carattere \*, tuttavia, il campo in ingresso corrispondente è ignorato: non viene fatto alcun assegnamento. Un campo in ingresso è definito come una stringa di caratteri che non siano spazi, e si estende fino allo spazio successivo o fin dove arriva, se specificata, l'ampiezza del campo. Questo significa che `scanf` potrà leggere i dati passando da una riga all'altra, perché i caratteri newline sono considerati spazi; appartengono a questa categoria lo spazio bianco ordinario, il carattere di tabulazione, il carattere newline, il ritorno del carrello, il carattere di tabulazione verticale e il salto pagina ("formfeed").

Il carattere di conversione stabilisce l'interpretazione da dare al campo in ingresso. L'argomento corrispondente non può che essere un puntatore, come prescritto dalla semantica della chiamata per valore del C. I caratteri di conversione sono illustrati nella Tabella 7.2.

I caratteri di conversione d, i, o, u e x possono essere preceduti da h per indicare che nella lista degli argomenti è presente un puntatore a short anziché a int, o da l (lettera elle), per segnalare che un puntatore a long compare nella lista degli argomenti. Similmente, i caratteri di conversione e, f, e g possono essere preceduti da l per indicare che la lista di argomenti contiene un puntatore a double anziché a float.

Per iniziare, si può rielaborare l'esempio della calcolatrice rudimentale del Capitolo 4, con l'aggiunta di `scanf` per la conversione dei dati in ingresso:

```
#include <stdio.h>

main() /* calcolatrice rudimentale */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

**Tabella 7.2** Conversioni principali di `scanf`.

CARATTERE	DATI IN INGRESSO; TIPO DELL'ARGOMENTO
d	intero in notazione decimale; int *
i	intero; int *. L'intero può essere in notazione ottale (segnalata da uno zero in testa) o esadecimale (segnalata da 0x o 0X in testa).
o	intero in notazione ottale (con o senza lo zero in testa); unsigned int *.
u	intero in notazione decimale privo di segno; unsigned int *.
x	intero in notazione esadecimale (con o senza 0x o 0X in testa); unsigned int *.
c	caratteri; char *. I successivi caratteri in ingresso (di default, il successivo) sono memorizzati nel punto indicato dall'argomento. Anche gli spazi contano come caratteri: per leggere il carattere successivo che non sia uno spazio, si usi %1s.
s	stringa di caratteri (non fra virgolette); char *, che punta a un vettore di caratteri di lunghezza sufficiente a contenere la stringa, più il carattere di terminazione aggiunto '\0'.
e, f, g	numero con virgola mobile e segno facoltativo, punto decimale facoltativo, esponente facoltativo; float *.
%	occorrenza letterale di % in ingresso; non ha luogo alcun assegnamento.

Poniamo di voler leggere righe in ingresso contenenti date in questa forma:

25 Dic 1988

L'istruzione `scanf` è

```
int day, year;
char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

Nessun & viene usato con `monthname`, dato che il nome di un vettore è un puntatore.

Caratteri che non specifichino conversioni possono apparire nella stringa di composizione di `scanf`, ma devono corrispondere letteralmente ai medesimi caratteri in ingresso. Quindi potremmo leggere date della forma mm/gg/aa con questa istruzione `scanf`:

```
int day, month, year;

scanf("%d/%d/%d", &month, &day, &year);
```

La funzione `scanf` ignora gli spazi (spazi ordinari, caratteri di tabulazione, caratteri newline e così via) durante la lettura dei dati in ingresso. Per quanto riguarda i dati il cui formato non è predeterminato, spesso il modo migliore è leggere una riga per volta, per poi analizzarla con `sscanf`. Per esempio, volendo leggere righe che potrebbero contenere una data in una delle due forme viste prima, potremmo scrivere quanto segue.

```

while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("forma valida: %s\n", line); /* 25 Dic 1988 */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("forma valida: %s\n", line); /* della forma mm/gg/aa */
    else
        printf("forma non ammessa: %s\n", line); /* forma non ammessa */
}

```

Le chiamate a `scanf` possono essere mescolate con chiamate ad altre funzioni che leggono dati in ingresso. La successiva chiamata a una qualsiasi di queste funzioni comincerà leggendo il primo carattere non letto da `scanf`.

Una raccomandazione finale: gli argomenti di `scanf` e `sscanf` *devono* essere puntatori. L'errore di gran lunga più comune è scrivere

```
scanf("%d", n);
```

anziché

```
scanf("%d", &n);
```

Questo errore passa spesso inosservato in fase di compilazione.

**Esercizio 7.4** Si scriva una versione non ufficiale di `scanf` analoga a `minprintf` del paragrafo precedente.

**Esercizio 7.5** Si rielabori la calcolatrice in notazione polacca del Capitolo 4 cosicché impieghi `scanf` e/o `sscanf` per leggere e convertire i dati in ingresso.

## 7.5 Accesso ai file

Gli esempi fatti finora hanno sempre letto dallo standard input e scritto sullo standard output, che sono definiti automaticamente dal sistema operativo locale.

Il passo successivo è di scrivere del codice che acceda a un file *non* ancora correlato al codice stesso. Un programma che illustra la necessità di tali operazioni è `cat`, il quale concatena allo standard output una serie di file specificati. Esso è usato per visualizzare file sullo schermo, e serve anche da collettore generico dell'input per programmi che non sono in grado di accedere ai file per nome. Per esempio, il comando

```
cat x.c y.c
```

stampa i contenuti dei file `x.c` e `y.c` (e nulla di più) sullo standard output.

Il problema, però, è proprio quello di accedere ai file, ovvero come mettere in relazione i nomi esterni dei file, stabiliti da un utente, alle istruzioni che leggono i dati.

Le regole sono semplici: un file, prima di essere letto o modificato, deve essere *aperto* dalla funzione della libreria `fopen`, la quale accetta un nome esterno, come `x.c` o `y.c`, concorda alcune operazioni preliminari con il sistema operativo (i cui dettagli non è il caso di approfondire), e restituisce un puntatore da usare nelle susseguenti operazioni di lettura o scrittura di quel file.

Questo puntatore, chiamato il *puntatore al file*, punta a una struttura contenente informazioni sul file, quali l'ubicazione di un buffer, la posizione del carattere corrente nel buffer, se si stia leggendo dal file o scrivendo su di esso, se siano incorsi errori, o se sia stata raggiunta la fine del testo (`EOF`). L'utente può ignorare i dettagli, dal momento che le definizioni contenute in `<stdio.h>` comprendono la dichiarazione di una struttura chiamata `FILE`. L'unica dichiarazione richiesta per un puntatore a un file è esemplificata da

```

FILE *fp;
FILE *fopen(char *name, char *mode);

```

Essa dichiara che `fp` è un puntatore a `FILE`, e che `fopen` restituisce un puntatore a `FILE`. Si noti che `FILE` è un nome di tipo, come `int`, e non il contrassegno di una struttura, ed è definito tramite `typedef`. (Nel Paragrafo 8.5 sono contenuti particolari su come `fopen` può essere implementata nel sistema UNIX.)

La chiamata a `fopen` in un programma è

```
fp = fopen(name, mode);
```

Il primo argomento di `fopen` è una stringa di caratteri contenente il nome del file. Il secondo argomento, `mode` (cioè modalità di accesso), anch'esso una stringa di caratteri, chiarisce come si intende usare il file. Tra le modalità consentite vi sono la lettura ("r", per "read"), la scrittura ("w", per "write"), e l'aggiunta in coda ("a", per "append"). Alcuni sistemi distinguono tra file di testo e binari; per questi ultimi, è necessario che una "b" sia allegata in coda alla stringa `mode`.

Se si prova ad aprire un file inesistente per scrivere o aggiungere qualcosa, esso sarà creato ex-novo, se possibile. Aprire un file esistente per scrivere fa sì che i vecchi contenuti siano dismessi, mentre ciò non accade se lo si apre per fare un'aggiunta. Tentare di leggere da un file inesistente è un errore, ma si può incorrere anche in altre cause d'errore, come il tentativo di leggere un file senza autorizzazione. Nell'eventualità di un qualunque errore, `fopen` restituirà `NULL`. (L'errore può essere individuato con maggior precisione; si veda l'analisi delle funzioni per la gestione degli errori alla fine del Paragrafo B.1 nell'Appendice B.)

Dopo l'apertura del file, è necessario ricorrere a un metodo per la lettura o la scrittura. Fra le varie possibilità, `getc` e `putc` sono le più semplici. La funzione `getc` restituisce il carattere successivo proveniente da un file, e richiede il puntatore al file per sapere da dove leggere.

```
int getc(FILE *fp)
```

Essa restituisce il carattere successivo disponibile dal flusso in ingresso denotato da `fp`; restituisce `EOF` in caso di fine del testo o errore.

La funzione `putc` produce dati in uscita:

```
int putc(int c, FILE *fp)
```

Essa scrive il carattere `c` sul file `fp` e restituisce il carattere scritto, oppure `EOF` se c'è un errore. Come `getchar` e `putchar`, `getc` e `putc` possono essere implementate come macro anziché funzioni.

Quando si esegue un programma in C, il sistema operativo ha il compito di aprire tre file e di associarvi altrettanti puntatori. Si tratta dello standard input, dello standard output, e del cosiddetto standard error, un flusso per il trattamento degli errori; i rispettivi puntatori si chiamano `stdin`, `stdout` e `stderr`, e sono dichiarati in `<stdio.h>`. Di norma, `stdin` si riferisce alla tastiera, mentre `stdout` e `stderr` allo schermo, ma `stdin` e `stdout` possono essere rediretti secondo le tecniche descritte nel Paragrafo 7.1.

Le funzioni `getchar` e `putchar` si possono definire in termini di `getc`, `putc`, `stdin` e `stdout` come segue:

```
#define getchar()    getc(stdin)
#define putchar(c)   putc((c), stdout)
```

Per la formattazione dei dati in ingresso e uscita da e su file, possono essere usate le funzioni `fscanf` e `fprintf`: sono identiche a `scanf` e `printf`, salvo che il primo argomento è un puntatore a un file che specifica il file da cui leggere o su cui scrivere; la stringa di composizione è il secondo argomento (`format`).

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

Avendo approntato i preliminari, siamo ora in condizione di scrivere il programma `cat` per concatenare file. L'impianto che adotteremo si è dimostrato efficiente in molti programmi: se vi sono argomenti dalla riga di comando, essi sono interpretati come nomi di file, ed elaborati nell'ordine; se non ci sono argomenti, si elabora lo standard input.

```
#include <stdio.h>

/* cat: concatena dei file, versione 1 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);

    if (argc == 1) /* niente argomenti; copia lo standard input */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(++argv, "r")) == NULL) {
                printf("cat: non riesco ad aprire %s\n", *argv);
                return 1;
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    return 0;
}

/* filecopy: copia il file ifp sul file ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;
```

```
    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}
```

I puntatori a file `stdin` e `stdout` sono oggetti di tipo `FILE *`. Si tratta, tuttavia, di costanti, non di variabili, motivo per cui non è possibile assegnare loro alcunché.

La funzione

```
int fclose(FILE *fp)
```

è l'inverso di `fopen`: rimuove l'associazione tra il puntatore a file e il suo nome esterno, instaurata in precedenza da `fopen`, liberando il puntatore a file per un altro file. Dato che quasi tutti i sistemi operativi pongono un limite al numero di file che un programma può tenere aperti allo stesso tempo, è una buona idea liberare i puntatori a file quando non servono più, come abbiamo fatto in `cat`. C'è anche un'altra ragione che rende utile l'applicazione di `fclose` a un file in uscita: essa svuota il buffer nel quale `putc` sta raccogliendo i dati in uscita, completando la loro effettiva scrittura sul file. La funzione `fclose` è invocata automaticamente per ciascun file aperto quando un programma termina in maniera normale. (È possibile chiudere `stdin` e `stdout` se non sono necessari, e riassegnarli nuovamente con la funzione della libreria `freopen`.)

## 7.6 Gestione degli errori: `stderr` ed `exit`

Il trattamento che `cat` riserva agli errori non è ideale. Il problema sorge quando, per qualche motivo, non si possa accedere a uno dei file, nel qual caso l'analisi degli errori è stampata alla fine dell'output concatenato. La cosa è accettabile se i dati in uscita sono destinati allo schermo, ma non se il flusso finisce in un file oppure in un altro programma attraverso una pipe.

Per gestire meglio questa situazione, un secondo flusso in uscita, detto `stderr` (da standard error), è assegnato a un programma allo stesso modo di `stdin` e `stdout`. L'output prodotto su `stderr` appare normalmente sullo schermo, anche laddove lo standard output sia stato rediretto.

Riformuliamo `cat` in modo che scriva i suoi messaggi d'errore sullo standard error.

```
#include <stdio.h>

/* cat: concatena dei file, versione 2 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog = argv[0]; /* il nome del programma per gli errori */

    if (argc == 1) /* niente argomenti; copia lo standard input */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(++argv, "r")) == NULL) {
```

```

        fprintf(stderr, "%s: non riesco ad aprire %s\n",
            prog, *argv);
        exit(1);
    } else {
        filecopy(fp, stdout);
        fclose(fp);
    }
    if (ferror(stdout)) {
        fprintf(stderr, "%s: errore nello scrivere su stdout\n", prog);
        exit(2);
    }
    exit(0);
}

```

Il programma segnala gli errori in due modi. In primo luogo, l'analisi degli errori, prodotta da `fprintf`, è convogliata sullo `stderr`, quindi raggiunge lo schermo invece di scomparire in una pipe o in un file in uscita. Nel messaggio, abbiamo incluso il nome del programma, tratto da `argv[0]`: se il programma è usato insieme ad altri, viene chiarita l'origine di un eventuale errore.

In secondo luogo, il programma adopera la funzione `exit` della libreria standard che, una volta chiamata, ne termina l'esecuzione. L'argomento di `exit` è a disposizione di qualsivoglia processo abbia invocato il programma, e dunque la riuscita o il fallimento della sua esecuzione possono essere rilevati da un altro programma che usi il nostro come processo secondario. Per convenzione, un valore di ritorno pari a zero segnala che tutto è andato bene; valori diversi da zero segnalano di solito situazioni anomale. La funzione `exit` chiama `fclose` per ogni file in uscita aperto, al fine di convogliare nel file l'eventuale output residuo nel buffer.

All'interno di `main`, `return espr` è equivalente a `exit(espr)`; la seconda forma ha il vantaggio di poter essere usata da altre funzioni, e a tali chiamate si può risalire con un programma di ricerca di pattern, come quello del Capitolo 5.

La funzione `ferror` restituisce un valore non nullo se il flusso `fp` ha dato luogo a un errore.

```
int ferror(FILE *fp)
```

Malgrado siano rari, esistono errori provocati dalle operazioni di scrittura (si pensi, per esempio, a un disco pieno), per cui un programma che produca dati in uscita dovrebbe tenere sotto controllo anche questo aspetto.

La funzione `feof(FILE *)` è analoga a `ferror`; essa restituisce un valore non nullo qualora si sia riscontrata la fine del file specificato.

```
int feof(FILE *fp)
```

Non si è attribuita grande importanza allo stato di terminazione dei brevi programmi di carattere illustrativo esaminati in questo libro, ma un programma serio dovrebbe garantire all'ambiente ospitante la restituzione di valori ragionevoli e utili.

## 7.7 Input e output delle righe

La libreria standard mette a disposizione una funzione per l'acquisizione dei dati, detta `fgets`, simile alla funzione `getline` usata in capitoli precedenti:

```
char *fgets(char *line, int maxline, FILE *fp)
```

Essa legge la riga in ingresso successiva (compreso il carattere newline) dal file `fp`, e la memorizza nel vettore di caratteri `line`; sarà letto un numero di caratteri non superiore a `maxline-1`. La riga che ne risulta termina con '\0'. Normalmente `fgets` restituisce `line`; alla fine del testo o in caso d'errore restituisce `NULL`. (La nostra `getline` restituisce la lunghezza della riga, che è un valore più utile; zero significa fine del file.)

Quanto ai dati in uscita, la funzione `fputs` scrive una stringa (che non deve contenere un carattere newline) su un file:

```
int fputs(char *line, FILE *fp)
```

Essa restituisce `EOF` se si verifica un errore, e zero altrimenti.

Le funzioni della libreria `gets` e `puts` sono simili a `fgets` e `fputs`, ma agiscono su `stdin` e `stdout`. Può confondere il fatto che `gets` cancella il '\n' finale, mentre `puts` lo aggiunge.

Per dimostrare che funzioni del genere non hanno nulla di speciale, presentiamo qui `fgets` e `fputs`, copiate dalla libreria standard del nostro sistema:<sup>(1)</sup>

```

/* fgets: get at most n chars from iop */
/* fgets: legge al massimo n caratteri da iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}

/* fputs: put string s on file iop */
/* fputs: scrive la stringa s sul file iop */
int fputs(char *s, FILE *iop)
{
    int c;

```

<sup>(1)</sup> N.d.R. In questo caso sono stati mantenuti i commenti originali in inglese, poiché gli Autori citano una libreria standard alla lettera. La traduzione italiana viene riportata per comodità del lettore. Si tenga presente che nell'errata corrigente gli Autori annotano: "Testo e codice del Paragrafo 7.7: la funzione `fputs` restituisce `EOF` in caso d'errore, e un valore non negativo altrimenti."

```

while (c = *s++)
    putc(c, iop);
return ferror(iop) ? EOF : 0;
}

```

Lo standard stabilisce che, in caso d'errore, `ferror` restituisca un valore non nullo; `fputs` restituisce `EOF` in caso d'errore, e un valore non negativo altrimenti.

È facile implementare la nostra `getline` partendo da `fgets`:

```

/* getline: legge una riga in ingresso, ne restituisce la lunghezza */
int getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
        return 0;
    else
        return strlen(line);
}

```

**Esercizio 7.6** Si scriva un programma per il confronto di due file, che visualizzi la prima riga dove divergono.

**Esercizio 7.7** Si rielabori il programma del Capitolo 5 per la ricerca di un pattern in un testo, in modo che esegua la ricerca su un insieme di file specificati dalla riga di comando o, in assenza di tali argomenti, sullo standard input. Si dovrebbe visualizzare il nome del file quando si incontra una riga in cui compare la forma cercata?

**Esercizio 7.8** Si scriva un programma per la stampa di una serie di file, ciascuno con inizio su una pagina nuova; si stampino anche i titoli dei file e i numeri di pagina.

## 7.8 Funzioni varie

La libreria standard fornisce una vasta gamma di funzioni. Questo paragrafo offre un quadro riassuntivo delle più utili. Ulteriori dettagli e molte altre funzioni si possono consultare nell'Appendice B.

### 7.8.1 Operazioni sulle stringhe

Abbiamo già menzionato le funzioni `strlen`, `strcpy`, `strcat` e `strcmp` per manipolare le stringhe, che si trovano in `<string.h>`. Nell'elenco seguente, `s` e `t` sono di tipo `char *`, mentre `c` e `n` sono interi.

<code>strcat(s, t)</code>	concatena <code>t</code> alla fine di <code>s</code>
<code>strncat(s, t, n)</code>	concatena <code>n</code> caratteri di <code>t</code> alla fine di <code>s</code>
<code>strcmp(s, t)</code>	restituisce valore negativo, zero o positivo per <code>s &lt; t</code> , <code>s == t</code> o <code>s &gt; t</code>
<code>strncmp(s, t, n)</code>	come <code>strcmp</code> ma solo per i primi <code>n</code> caratteri
<code>strcpy(s, t)</code>	copia <code>t</code> in <code>s</code>
<code>strncpy(s, t, n)</code>	copia un massimo di <code>n</code> caratteri di <code>t</code> in <code>s</code>

<code>strlen(s)</code>	restituisce la lunghezza di <code>s</code>
<code>strchr(s, c)</code>	restituisce un puntatore alla prima occorrenza di <code>c</code> in <code>s</code> , o <code>NULL</code> se <code>c</code> non compare in <code>s</code>
<code> strrchr(s, c)</code>	restituisce un puntatore all'ultima occorrenza di <code>c</code> in <code>s</code> , o <code>NULL</code> se <code>c</code> non compare in <code>s</code>

### 7.8.2 Verifiche e conversioni di classi di caratteri

Svariate funzioni di `<ctype.h>` effettuano conversioni e verifiche sui caratteri. Di seguito, `c` è un intero che può essere rappresentato come `unsigned char`, o `EOF`. Le funzioni restituiscono interi.

<code>isalpha(c)</code>	non nullo se <code>c</code> è un carattere alfabetico, 0 altrimenti
<code>isupper(c)</code>	non nullo se <code>c</code> è un carattere maiuscolo, 0 altrimenti
<code>islower(c)</code>	non nullo se <code>c</code> è un carattere minuscolo, 0 altrimenti
<code>isdigit(c)</code>	non nullo se <code>c</code> è una cifra, 0 altrimenti
<code>isalnum(c)</code>	non nullo se uno fra <code>isalpha(c)</code> e <code>isdigit(c)</code> è non nullo, 0 altrimenti
<code>isspace(c)</code>	non nullo se <code>c</code> è uno spazio, un carattere di tabulazione, un carattere newline o ritorno del carrello, un salto pagina, un carattere di tabulazione verticale
<code>toupper(c)</code>	restituisce <code>c</code> convertita in maiuscola
<code>tolower(c)</code>	restituisce <code>c</code> convertita in minuscola

### 7.8.3 Ungetc

La libreria standard fornisce una versione piuttosto limitata della funzione `ungetch` descritta nel Capitolo 4: si chiama `ungetc`.

```
int ungetc(int c, FILE *fp)
```

riporta il carattere `c` di nuovo sul file `fp`, e restituisce `c`, oppure `EOF` in caso d'errore. Per ogni file è garantito l'accantonamento di un solo carattere; `ungetc` può essere usata con tutte le funzioni per la gestione dei dati in ingresso, come `scanf`, `getc` o `getchar`.

### 7.8.4 Esecuzione dei comandi

La funzione `system(char *s)` chiede al sistema operativo ambiente l'esecuzione del comando contenuto nella stringa di caratteri `s`, quindi riprende l'esecuzione del programma in corso. I contenuti di `s` dipendono strettamente dal sistema operativo locale. Con un esempio elementare, sui sistemi UNIX, l'istruzione

```
system("date");
```

esegue il programma `date`, il quale stampa sullo standard output la data e l'ora del giorno. La funzione `system` restituisce, per conto del comando eseguito, il valore intero dello stato, dipendente dal sistema. In UNIX, il valore è quello restituito da `exit`.

### 7.8.5 Gestione della memoria

Le funzioni `malloc` e `calloc` ottengono dinamicamente blocchi di memoria.

```
void *malloc(size_t n)
```

restituisce un puntatore a  $n$  byte di memoria non inizializzata, ovvero `NULL` se la richiesta non può essere soddisfatta.

```
void *calloc(size_t n, size_t size)
```

restituisce un puntatore a uno spazio adeguato per un vettore di  $n$  oggetti della misura specificata, ovvero `NULL` se la richiesta non può essere soddisfatta. La memoria allocata è inizializzata a zero.

Il puntatore restituito da `malloc` o `calloc` è correttamente allineato per l'oggetto in questione, ma deve essere forzato nel tipo opportuno, come in<sup>(2)</sup>

```
int *ip;
ip = (int *) calloc(n, sizeof(int));
```

La chiamata `free(p)` libera lo spazio di memoria puntato da  $p$ , dove  $p$  originariamente proveniva da una chiamata a `malloc` o `calloc`. Non c'è un ordine di priorità tassativo nel liberare spazio in memoria, ma è un errore grossolano deallocare qualcosa che non sia stato ottenuto chiamando `calloc` o `malloc`.

Un altro errore è tentare di usare un'area di memoria dopo averla liberata. Un brano di codice tipico, ma sbagliato, è questo ciclo che libera gli elementi di una lista:

```
for (p = head; p != NULL; p = p->next) /* ERRATO */
    free(p);
```

La procedura corretta è di salvare tutto il necessario prima di liberare alcunché:

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
}
```

Nel Paragrafo 8.7 è illustrata l'implementazione di un allocatore di memoria simile a `malloc`, in cui si possono liberare in qualunque ordine i blocchi già allocati.

### 7.8.6 Funzioni matematiche

In `<math.h>` sono dichiarate più di venti funzioni matematiche, e di seguito sono elencate alcune di quelle usate più spesso. Ognuna impiega uno o due argomenti `double`, e restituisce un valore di tipo `double`.

<code>sin(x)</code>	seno di $x$ , $x$ in radianti
<code>cos(x)</code>	coseno di $x$ , $x$ in radianti
<code>atan2(y,x)</code>	arcotangente di $y/x$ , in radianti
<code>exp(x)</code>	funzione esponenziale $e^x$
<code>log(x)</code>	logaritmo naturale (in base $e$ ) di $x$ ( $x > 0$ )
<code>log10(x)</code>	logaritmo comune (in base 10) di $x$ ( $x > 0$ )
<code>pow(x,y)</code>	$x^y$
<code>sqrt(x)</code>	radice quadrata di $x$ ( $x \geq 0$ )
<code>fabs(x)</code>	valore assoluto di $x$

### 7.8.7 Generazione di numeri casuali

La funzione `rand()` computa una sequenza di interi pseudo-casuali nell'intervallo tra zero e `RAND_MAX`, che è definito in `<stdlib.h>`. Un modo di produrre numeri casuali con virgola mobile maggiori di 0 uguali a zero ma minori di uno è

```
#define frand() ((double) rand() / (RAND_MAX+1.0))
```

(Se il lettore dispone di una libreria con un generatore di numeri psuedo-casuali con virgola mobile, è probabile che esso abbia proprietà statistiche migliori di questo.)

La funzione `srand(unsigned)` imposta il seme di `rand`. L'implementazione portabile di `rand` e `srand` consigliata dallo standard compare nel Paragrafo 2.7.

**Esercizio 7.9** Funzioni come `isupper` possono essere implementate privilegiando l'efficienza in termini di spazio di memoria o di tempo d'esecuzione. Si esplorino entrambe le possibilità.

2. N.d.R. Questo passo è obsoleto. Si veda la seconda nota a piè di pagina del Paragrafo 6.5, tratta dall'errata corrigé al testo pubblicata elettronicamente.

## Interfaccia con il sistema UNIX

**A**I SERVIZI OFFERTI DAL SISTEMA OPERATIVO UNIX si accede mediante una serie di *chiamate di sistema* (*system call*), che sono in realtà funzioni interne al sistema invocabili da programmi dell'utente. In questo capitolo è illustrato l'uso da programmi C di alcune delle chiamate di sistema più importanti. Per l'utente di UNIX, ciò dovrebbe costituire un aiuto immediato, dato che talvolta è necessario sfruttare le chiamate di sistema per raggiungere la massima efficienza, o per accedere a qualche funzionalità non presente nella libreria. Anche chi usa il C con un sistema operativo diverso, comunque, dovrebbe essere in grado di trarre profitto dallo studio di questi esempi; nonostante possano variare i particolari, il codice è simile per qualsiasi sistema. Visto che la libreria del C ANSI è modellata in molti casi sulle funzionalità dell'ambiente UNIX, il codice dovrebbe facilitare anche la comprensione della libreria.

Il capitolo è suddiviso in tre parti: input e output, file system e allocazione di memoria. Le prime due parti presuppongono una modesta familiarità con le caratteristiche esterne dei sistemi UNIX.

Nel Capitolo 7 è stata analizzata un'interfaccia per l'ingresso e l'uscita dei dati uniforme e indipendente dai vari sistemi operativi. In base al sistema specifico, l'implementazione della libreria standard deve essere adattata alle funzionalità fornite dal sistema ambiente. Nei paragrafi successivi saranno descritte le chiamate di sistema relative a input e output, mostrando come, grazie a esse, si possano implementare alcune parti della libreria standard.

## 8.1 Descrittori dei file

Nel sistema operativo UNIX, tutto ciò che riguarda l'ingresso e l'uscita dei dati avviene tramite la lettura o la scrittura di file, poiché tutti i dispositivi periferici, persino la tastiera e il video, sono considerati alla stregua di file appartenenti al file system. Questo significa che un'unica interfaccia omogenea si occupa della comunicazione tra un programma e i dispositivi periferici.

Nel caso più generale, prima di leggere o scrivere da o su un file, bisogna avvertire il sistema di questa intenzione, un processo chiamato *apertura* del file. Se si è in procinto di scrivere su un file, può anche essere necessario crearlo o eliminarne i contenuti precedenti. Il sistema controlla che si abbia diritto a farlo (Il file esiste? L'accesso è stato autorizzato?) e, se tutto è in regola, restituisce al programma un piccolo intero non negativo, chiamato *descrittore del file*. Da questo momento, tutti i riferimenti al file avvengono tramite il suo descrittore. (Il descrittore corrisponde al puntatore a file usato dalla libreria standard, o al file handle di MS-DOS.) Tutte le informazioni riguardanti un file aperto sono curate dal sistema; il programma utente si riferisce al file unicamente attraverso il suo descrittore.

Dato che l'ingresso e l'uscita dei dati coinvolgono spesso tastiera e video, sono invalse alcune prassi per renderne agevole l'utilizzo. Quando l'interprete dei comandi (detto "shell") esegue un programma, vengono aperti tre file, i cui descrittori sono rispettivamente 0, 1 e 2, denominati standard input, standard output e standard error. Se un programma legge solo da 0 e scrive solo su 1 e 2, può eseguire le operazioni di ingresso e uscita dei dati senza preoccuparsi dell'apertura di alcun file.

L'utente di un programma può attuare la redirezione dell'I/O verso e da file con < e >:

```
prog <infile>>outfile
```

In questo caso, l'interprete assegna i descrittori 0 e 1 ai due file, *infile* e *outfile*. Normalmente il descrittore di 2 rimane associato al video, cosicché i messaggi d'errore possano essere visualizzati. Considerazioni simili valgono per il meccanismo della pipe. In ogni caso, le associazioni fra descrittori e file sono modificate dall'interprete, non dal programma, che non conosce la provenienza dei suoi dati in ingresso né la destinazione di quelli in uscita, sempre che continui a impiegare il file 0 per l'input e i file 1 e 2 per l'output.

## 8.2 I/O a basso livello: read e write

L'input e l'output avvengono tramite le chiamate di sistema *read* e *write*, a cui si accede dai programmi in C tramite le funzioni omonime *read* e *write*. Per entrambe, il primo argomento è un descrittore del file, il secondo un vettore di caratteri appartenente al programma interessato dall'ingresso o dall'uscita dei dati, e il terzo è il numero di byte da trasferire.

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

Ogni chiamata restituisce il numero di byte trasferiti. In lettura, il numero restituito può anche essere minore del numero richiesto: 0 implica la fine del file, mentre -1 denota un erro-

re di qualche tipo. Quanto alla scrittura, il valore restituito è dato dal numero di byte scritti; se questo numero non è pari a quello richiesto, si è incorsi in un errore.

Con una singola chiamata si può leggere o scrivere qualunque numero di byte. I valori più diffusi sono 1, cioè un carattere per volta ("unbuffered", ovvero senza uso di un buffer), e un numero come 1024 o 4096 che corrisponde alla dimensione di un blocco su un dispositivo periferico. Dimensioni maggiori si rivelano più efficienti, poiché consentono di diminuire il numero delle chiamate di sistema.

Per mettere a frutto queste informazioni, possiamo scrivere un semplice programma che replica in uscita i propri dati in ingresso, l'equivalente di quanto fatto nel Capitolo 1. Il programma permette di copiare da e verso qualunque cosa, dal momento che l'input e l'output possono essere rediretti da e verso qualsiasi file o dispositivo periferico.

```
#include "syscalls.h"

main() /* replica in uscita i propri dati in ingresso */
{
    char buf[BUFSIZ];
    int n;

    while ((n = read(0, buf, BUFSIZ)) > 0)
        write(1, buf, n);
    return 0;
}
```

Abbiamo raccolto i prototipi delle funzioni per le chiamate di sistema in un file chiamato *syscalls.h* al fine di poterlo riutilizzare per i programmi di questo capitolo; tuttavia, questo nome non è standard.

Il parametro *BUFSIZ* (da "buffer size"), definito in *syscalls.h*, è una grandezza adatta al sistema locale. Se la dimensione del file non è un multiplo di *BUFSIZ*, alcune operazioni *read* trasmetteranno alle operazioni *write* un numero di byte da scrivere inferiore a *BUFSIZ*; la chiamata a *read* successiva a queste restituirà zero.

È istruttivo studiare come *read* e *write* possano servire a costruire funzioni di più alto livello come *getchar*, *putchar* e così via. Per esempio, ecco una versione di *getchar* che legge dati in ingresso senza usare un buffer, leggendo dallo standard input carattere per carattere.

```
#include "syscalls.h"

/* getchar: lettura di caratteri singoli senza buffer */
int getchar(void)
{
    char c;

    return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
}
```

La variabile *c* deve essere di tipo *char*, perché la funzione *read* richiede un puntatore a carattere. Forzando *c* nel tipo *unsigned char* all'ultima istruzione, si elimina ogni problema legato all'estensione del segno.

La seconda stesura di getchar trasferisce un grosso blocco di dati in memoria, restituendo poi al chiamante un carattere per volta.

```
#include "syscalls.h"

/* getchar: semplice versione con buffer */
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0) { /* il buffer è vuoto */
        n = read(0, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

Se si dovessero compilare queste versioni di getchar in presenza dell'intestazione `<stdio.h>`, sarebbe necessario eseguire la direttiva `#undef` per getchar, nel caso la funzione standard sia implementata come una macroistruzione.

### 8.3 Open, creat, close e unlink

Eccezion fatta per standard input, output ed error, i file da leggere o scrivere devono essere aperti esplicitamente. A tal fine sono previste due chiamate di sistema, open e creat (la parola completa sarebbe “create”).

La chiamata open è piuttosto simile alla funzione fopen trattata nel Capitolo 7, tranne che, invece di un puntatore a un file, restituisce un descrittore del file, che è un semplice intero. Essa restituisce -1 se si verifica un errore.

```
#include <fcntl.h>

int fd;
int open(char *name, int flags, int perms);

fd = open(name, flags, perms);
```

Come per fopen, l'argomento name è una stringa di caratteri contenente il nome del file. Il secondo argomento, flags, è un intero che precisa la modalità d'apertura del file; i valori più importanti sono

<code>O_RDONLY</code>	sola lettura
<code>O_WRONLY</code>	sola scrittura
<code>O_RDWR</code>	sia lettura che scrittura

Queste costanti sono definite in `<fcntl.h>` sui sistemi UNIX System V, e in `<sys/file.h>` sulle versioni Berkeley (BSD).

Per aprire un file esistente per la sola lettura,

```
fd = open(name, O_RDONLY, 0);
```

L'argomento perms (“permissions”) è sempre zero per gli utilizzi di open che prenderemo in esame.

È un errore tentare di aprire con open un file inesistente: la chiamata di sistema creat serve per creare nuovi file, o per riscrivere sopra i vecchi. Nel brano

```
int creat(char *name, int perms);
fd = creat(name, perms);
```

la chiamata di creat restituisce un descrittore del file se la creazione del file è andata a buon fine, e -1 altrimenti. Qualora il file esistesse già, creat tronca la sua lunghezza a zero, eliminandone i contenuti; non è quindi un errore applicare creat a un file esistente.

Se il file non esiste ancora, creat lo costruisce in conformità alle autorizzazioni specificate dall'argomento perms. Nel file system di UNIX vi sono nove bit di informazioni sulle autorizzazioni, che regolamentano l'accesso al file e le sue modalità (lettura, scrittura, esecuzione) da parte del proprietario, del gruppo da cui dipende il proprietario e di tutti gli altri utenti. Ne segue che un numero in notazione ottale a tre cifre è adatto per codificare le autorizzazioni. Per esempio, 0755 stabilisce che il proprietario è autorizzato alla lettura, alla scrittura e all'esecuzione, mentre tutti gli altri possono solo leggere ed eseguire il file.

Per capire meglio, si consideri una versione semplificata del comando cp di UNIX, che copia un file su un altro. Il nostro programma copia soltanto un file per volta, non permette che il secondo argomento sia una directory, e inventa le autorizzazioni invece di copiarle.

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666 /* autorizzazione RW per il proprietario, */
                  /* il gruppo e gli altri utenti */

void error(char *, ...);

/* cp: copia f1 su f2 */
main(int argc, char *argv[])
{
    int f1, f2, n;
    char buf[BUFSIZ];

    if (argc != 3)
        error("Uso: cp da a");
    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
        error("cp: non riesco ad aprire %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("cp: non riesco a creare %s, con modalita' %03o",
              argv[2], PERMS);
    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: errore durante la scrittura sul file %s", argv[2]);
    return 0;
}
```

Il programma crea il file in uscita con autorizzazioni fisse pari a 0666. La chiamata di sistema `stat`, descritta nel Paragrafo 8.6, consente di determinare le modalità di accesso a un file esistente e di applicarle quindi alla copia.

Si noti che la funzione `error` è invocata con una lista di argomenti di lunghezza variabile, come succedeva per `printf`. L'implementazione di `error` dimostra come usare un altro membro della famiglia di `printf`. La funzione della libreria standard `vprintf` è analoga a `printf`, tranne che la lista di argomenti di lunghezza variabile è sostituita da un singolo argomento, inizializzato tramite la macro `va_start`. Allo stesso modo, `vfprintf` e `vsprintf` corrispondono a `fprintf` e `sprintf`.

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

/* error: visualizza un messaggio d'errore e termina */
void error(char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, "errore: ");
    vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    va_end(args);
    exit(1);
}
```

Il numero di file che un programma può tenere aperti in contemporanea è limitato, spesso attorno ai 20. Di conseguenza, qualunque programma che intenda elaborare molti file deve essere predisposto per riutilizzare i descrittori di file. La funzione `close(int fd)` spezza il legame tra il descrittore del file e il file aperto, e libera il descrittore in modo che possa essere nuovamente utilizzato; essa corrisponde alla funzione `fclose` della libreria standard, con la differenza che non c'è alcun buffer da svuotare. L'arresto di un programma causato da `exit` o dalla terminazione del programma principale (funzione `main`) determina la chiusura di tutti i file aperti.

La funzione `unlink(char *name)` elimina il file `name` dal file system, e corrisponde alla funzione `remove` della libreria standard.

**Esercizio 8.1** Si riscriva il programma `cat` del Capitolo 7 usando `read`, `write`, `open` e `close` al posto dei rispettivi equivalenti della libreria standard. Si faccia qualche esperimento per confrontare i tempi d'esecuzione delle due versioni.

## 8.4 Accesso casuale: lseek

Lettura e scrittura dei dati avvengono normalmente in modo sequenziale: ogni chiamata a `read` o `write` accede, nel file, alla posizione immediatamente successiva al punto d'accesso della chiamata precedente. Quando necessario, peraltro, un file può essere letto o scritto in

qualsiasi ordine arbitrario. La chiamata di sistema `lseek` offre la possibilità di spostarsi all'interno di un file senza leggere o scrivere dati:

```
long lseek(int fd, long offset, int origin);
```

imposta a `offset` la posizione corrente nel file il cui descrittore è `fd`, posizione in cui avrà luogo la successiva operazione di lettura o scrittura e che si intende relativa al punto specificato da `origin`; quest'ultimo parametro può assumere i valori 0, 1 o 2 per indicare, rispettivamente, che lo scostamento (`offset`) sarà misurato dall'inizio, dalla posizione corrente o dalla fine del file. Per esempio, per aggiungere qualcosa in coda a un file (come con la redirezione `>>` nell'interprete di UNIX, o nelle chiamate a `fopen` con parametro `"a"`), se ne cerchi la fine prima di scrivere:

```
lseek(fd, 0L, 2);
```

Per tornare all'inizio,

```
lseek(fd, 0L, 0);
```

Si noti l'argomento `0L`, che potrebbe anche essere scritto come `(long) 0`, o anche solamente `0` se `lseek` è dichiarata correttamente.

Con `lseek` è possibile trattare i file più o meno alla stregua di vettori potenzialmente molto grandi, con l'inconveniente di tempi d'accesso più lunghi. Per esempio, la seguente funzione legge un qualsivoglia numero di byte, a partire da un punto arbitrario del file. La funzione restituisce il numero di byte letti, o -1 per segnalare un errore.

```
#include "syscalls.h"

/* get: legge n byte a partire dalla posizione pos */
int get(int fd, long pos, char *buf, int n)
{
    if (lseek(fd, pos, 0) >= 0) /* trova la posizione pos */
        return read(fd, buf, n);
    else
        return -1;
}
```

Il valore restituito da `lseek` è un intero di tipo `long` che dà la nuova posizione corrente nel file, o vale -1 se si è verificato un errore. La funzione della libreria standard `fseek` è simile a `lseek`, eccetto che il primo argomento è un `FILE *` e che il valore restituito è diverso da zero se si è verificato un errore.

## 8.5 Esempio di implementazione di fopen e getc

Dimostriamo come integrare alcuni degli aspetti fin qui analizzati per mezzo di un'implementazione delle funzioni `fopen` e `getc` della libreria standard.

Si ricorderà che i file della libreria standard sono descritti da puntatori a file anziché da descrittori di file. Un puntatore a file è un puntatore a una struttura contenente diverse in-

formazioni: un puntatore a un buffer, che permette la lettura del file in segmenti di grandi dimensioni; il numero di caratteri rimasti nel buffer; un puntatore alla posizione corrente nel buffer; il descrittore del file; campi di bit che ragguaglino sulla modalità di accesso, gli errori verificatisi, e così via.

La struttura dati che descrive un file si trova in `<stdio.h>`, intestazione da allegare obbligatoriamente (tramite direttiva `#include`) a qualunque file sorgente che impieghi funzioni per l'ingresso o l'uscita dei dati della libreria standard; l'intestazione è anche acclusa alle funzioni della libreria. Nel seguente estratto di una tipica intestazione `<stdio.h>`, i nomi il cui utilizzo è riservato alle funzioni della libreria sono preceduti dal carattere underscore (`_`), per evitare la sovrapposizione con nomi definiti dagli utenti. Si tratta di una convenzione applicata da tutte le funzioni della libreria standard.<sup>(1)</sup>

```

#define NULL      0
#define EOF       (-1)
#define BUFSIZ    1024
#define OPEN_MAX  20 /* max #files open at once */
                  /* numero max di file apribili in contemporanea */

typedef struct _iobuf {
    int   cnt;        /* characters left */
                      /* caratteri rimasti (nel buffer) */

    char *ptr;       /* next character position */
                      /* prossima posizione (nel buffer) */

    char *base;      /* location of buffer */
                      /* indirizzo del buffer */

    int   flag;      /* mode of file access */
                      /* modalita' di accesso al file */

    int   fd;        /* file descriptor */
                      /* descrittore del file */

} FILE;
extern FILE _iob[OPEN_MAX];

#define stdin    (&_iob[0])
#define stdout   (&_iob[1])
#define stderr   (&_iob[2])

```

1. N.d.R. Poiché il seguente passo di codice intende illustrare i contenuti di una tipica intestazione `<stdio.h>`, abbiamo mantenuto i commenti in originale, riportandone la traduzione in italiano per comodità del lettore. Lo stesso vale per altri brani di codice riportati nel seguito di questo capitolo. Si tenga inoltre presente che, come segnalato dall'errata corrigere al testo pubblicata elettronicamente, il nome standard per la costante `OPEN_MAX` è in realtà `FOPEN_MAX`.

```

enum _flags {
    _READ   = 01,      /* file open for reading */
                      /* file aperto per la sola lettura */

    _WRITE  = 02,      /* file open for writing */
                      /* file aperto per la sola scrittura */

    _UNBUF  = 04,      /* file is unbuffered */
                      /* file privo di buffer */

    _EOF    = 010,     /* EOF has occurred on this file */
                      /* si e' riscontrata la fine del file (EOF) */

    _ERR    = 020,     /* error occurred on this file */
                      /* c'e' stato un errore relativo a questo file */
};

int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define feof(p)      (((p)->flag & _EOF) != 0)
#define ferror(p)    (((p)->flag & _ERR) != 0)
#define fileno(p)    ((p)->fd)

#define getc(p)       (--(p)->cnt >= 0 \
                     ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define putc(x,p)    (--(p)->cnt >= 0 \
                     ? *(p)->ptr++ = (x) : _flushbuf((x),p))

#define getchar()    getc(stdin)
#define putchar(x)   putc((x), stdout)

```

La macro `getc`, normalmente, decrementa il contatore, fa avanzare il puntatore e restituisce il carattere. (Va ricordato che, nelle direttive `#define` più lunghe di una riga, la continuazione su un'altra riga si deve segnalare con una barra inversa, cioè `\`). Se il contatore diventa negativo, tuttavia, `getc` chiama la funzione `_fillbuf` per rifornire il buffer, reimpostare i contenuti della struttura e restituire un carattere. I caratteri restituiti saranno tutti di tipo `unsigned char`, il che assicura che avranno valore positivo.

Anche se non ci addentreremo nei dettagli, è stata inclusa la definizione di `putc` per dimostrare che essa opera in maniera estremamente simile a `getc`, chiamando una funzione `_flushbuf` quando il suo buffer è pieno. Abbiamo incluso anche le macro per l'accesso ai bit che segnalano un errore o la fine del file, e per l'accesso al descrittore del file.

Possiamo ora scrivere la funzione `fopen`, che si occupa prevalentemente di aprire il file, impostarne opportunamente la posizione corrente e attivare i bit appropriati per segnalare lo stato del file. Il codice di `fopen` non alloca spazio di memoria per il buffer; a ciò provvede `_fillbuf` durante la prima lettura del file.

```

#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666 /* autorizzazione RW per il proprietario, */
/* il gruppo e gli altri utenti */

/* fopen: apre il file, restituisce un puntatore al file */
FILE *fopen(char *name, char *mode)
{
    int fd;
    FILE *fp;

    if (*mode != 'r' && *mode != 'w' && *mode != 'a')
        return NULL;
    for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
        if ((fp->flag & (_READ | _WRITE)) == 0)
            break; /* c'e' spazio per aprire ancora un file */
    if (fp >= _iob + OPEN_MAX) /* non c'e' piu' spazio */
        return NULL;

    if (*mode == 'w')
        fd = creat(name, PERMS);
    else if (*mode == 'a') {
        if ((fd = open(name, O_WRONLY, 0)) == -1)
            fd = creat(name, PERMS);
        lseek(fd, 0L, 2);
    } else
        fd = open(name, O_RDONLY, 0);
    if (fd == -1) /* non riesce ad accedere al file name */
        return NULL;
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag = (*mode == 'r') ? _READ : _WRITE;
    return fp;
}

```

In questa versione, `fopen` non esaurisce tutte le modalità di accesso previste dallo standard, sebbene aggiungerle non richiederebbe molte altre istruzioni. In particolare, la nostra `fopen` non riconosce la “`b`” che sta per accesso binario, data l’inutilità dell’opzione sui sistemi UNIX, e nemmeno il “`+`” che permette sia la lettura che la scrittura.

La prima chiamata a `getc` per un file specifico trova il contatore dei caratteri nel buffer a zero, cosa che impone di chiamare `_fillbuf`. Se `_fillbuf` accetta che il file non è aperto per la lettura, restituisce subito `EOF`. Diversamente, tenta di allocare un buffer, posto che la modalità di lettura preveda l’uso di un buffer.

Una volta ottenuto il buffer, `_fillbuf` chiama `read` per riempirlo, imposta il contatore e i puntatori e restituisce il carattere all’inizio del buffer. Le chiamate successive a `_fillbuf` troveranno un buffer già allocato.

```

#include "syscalls.h"

/* _fillbuf: alloca e rifornisce il buffer */
int _fillbuf(FILE *fp)
{
    int bufsize;

    if ((fp->flag & (_READ | _EOF | _ERR)) != _READ)
        return EOF;
    bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
    if (fp->base == NULL) /* buffer non ancora allocato */
        if ((fp->base = (char *) malloc(bufsize)) == NULL)
            return EOF; /* non riesce ad allocare il buffer */
    fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, bufsize);
    if (--fp->cnt < 0) {
        if (fp->cnt == -1)
            fp->flag |= _EOF;
        else
            fp->flag |= _ERR;
        fp->cnt = 0;
        return EOF;
    }
    return (unsigned char) *fp->ptr++;
}

```

L’unico punto ancora da affrontare è il meccanismo che dà avvio a tutto il resto. Il vettore `_iob` deve essere definito e inizializzato in rapporto a `stdin`, `stdout` e `stderr`:

```

FILE _iob[OPEN_MAX] = { /* stdin, stdout, stderr: */
{ 0, (char *) 0, (char *) 0, _READ, 0 },
{ 0, (char *) 0, (char *) 0, _WRITE, 1 },
{ 0, (char *) 0, (char *) 0, _WRITE | _UNBUF, 2 }
};

```

L’inizializzazione dei membri `flag` della struttura mostra che `stdin` sarà disponibile per la lettura, `stdout` per la scrittura e `stderr` per la scrittura senza l’uso di un buffer.

**Esercizio 8.2** Si riscrivano `fopen` e `_fillbuf` sfruttando campi di bit al posto di operazioni esplicite sui bit. Si mettano a confronto le dimensioni del codice e la velocità di esecuzione delle due versioni.

**Esercizio 8.3** Si progettino e si scrivano `_flushbuf`, `fflush` e `fclose`.

**Esercizio 8.4** La funzione della libreria standard

```
int fseek(FILE *fp, long offset, int origin)
```

è identica a `lseek`, con due eccezioni: `fp`, che è un puntatore a file anziché un descrittore di file, e il valore restituito, che è un intero relativo allo stato e non una posizione. Si scriva `fseek` e ci si assicuri che la funzione si coordini appropriatamente con l’impiego del buffer da parte delle altre funzioni della libreria.

## 8.6 Esempio di liste di directory

A volte è necessario determinare delle informazioni rispetto al file, e non al suo contenuto; si tratta di un modo diverso di interagire con il sistema. Un esempio calzante è dato dal comando **UNIX ls**, un programma che elenca le directory: esso visualizza i nomi dei file in una directory e, a discrezione dell'utente, altre informazioni quali le dimensioni, le autorizzazioni e così via. Il comando **dir** di MS-DOS è analogo.

Dal momento che una directory di UNIX è semplicemente un file, **ls** non deve fare altro che leggerlo per acquisire i nomi dei file in esso contenuti. Per ottenere altre informazioni su un file, come la sua grandezza, è però d'obbligo usare una chiamata di sistema. In altri ambienti potrebbe servire una chiamata di sistema anche per accedere ai nomi dei file; ed è questo, per esempio, il caso per MS-DOS. Il nostro scopo è di fornire accesso alle informazioni con una modalità relativamente indipendente dal sistema, anche se l'implementazione potrà avere un alto grado di dipendenza da esso.

Illustreremo la questione con un programma chiamato **fsize**, una forma particolare di **ls**, che visualizza le dimensioni di tutti i file nominati nella propria lista di argomenti proveniente dalla riga di comando. Se uno dei file è una directory, **fsize** applica se stesso ricorsivamente a tale directory; in assenza di argomenti, elabora la directory corrente.

Per iniziare, passiamo brevemente in rassegna la struttura del file system di UNIX. Una *directory* è un file che contiene una lista di nomi di file, insieme ad alcune indicazioni sulla loro posizione. La "posizione" non è altro che un indice all'interno di un'altra tabella, detta "lista degli inode". Lo *inode* di un file è il luogo dove sono conservate tutte le informazioni relative al file, tranne il suo nome. Una directory consiste in genere di soli due elementi, il suo nome e un numero di inode.

Purtroppo, il formato e i contenuti specifici di una directory non sono i medesimi per ogni versione del sistema. Separiamo allora la procedura in due fasi, con l'intento di isolare le parti non portabili. Definiremo una struttura chiamata **Dirent** (da "Directory entry"), e tre funzioni, **opendir**, **readdir** e **closedir**, che consentono un accesso indipendente dal sistema al nome e al numero di inode di una directory. Scriveremo **fsize** sulla scia di questa interfaccia. Quindi mostreremo come implementare le suddette funzioni sui sistemi UNIX Version 7 e System V, e lasceremo le varianti come esercizi.

La struttura **Dirent** contiene il numero di inode e il nome. La lunghezza massima che può avere un nome di file contenuto nella directory è data da **NAME\_MAX**, che è un valore dipendente dal sistema. La funzione **opendir** restituisce un puntatore a una struttura chiamata **DIR**, analoga a **FILE**, impiegata da **readdir** e **closedir**. Tutte queste informazioni sono custodite in un file chiamato **dirent.h**.

```
#define NAME_MAX 14 /* massima lunghezza di un nome; */
                   /* dipendente dal sistema */

typedef struct { /* descrizione portabile della directory: */
    long ino;          /* numero inode */
    char name[NAME_MAX+1]; /* nome + '\0' terminale */
} Drent;
```

```
typedef struct { /* DIR minima: niente buffer, ecc. */
    int fd;           /* descrittore del file directory */
    Drent d;         /* la struttura che descrive la directory */
} DIR;
```

```
DIR *opendir(char *dirname);
Drent *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

La chiamata di sistema **stat** ha come argomento il nome di un file e restituisce le informazioni complete contenute nell'inode relativo al file, o -1 se c'è un errore. In altri termini,

```
char *name;
struct stat stbuf;
int stat(char *, struct stat *);

stat(name, &stbuf);
```

riempie la struttura **stbuf** di informazioni contenute nell'inode del file **name**. La struttura che descrive il valore restituito da **stat** è all'interno di **<sys/stat.h>**, e si presenta solitamente così:

```
struct stat /* inode information returned by stat */
/* informazioni relative all'inode restituite da stat */
{
    dev_t     st_dev;   /* device of inode */
                      /* dispositivo dell'inode */

    ino_t     st_ino;   /* inode number */
                      /* numero dell'inode */

    short    st_mode;  /* mode bits */
                      /* bit di modalita' */

    short    st_nlink; /* number of links to file */
                      /* numero di collegamenti al file */

    short    st_uid;   /* owner's user id */
                      /* identificatore del proprietario */

    short    st_gid;   /* owner's group id */
                      /* identificatore del gruppo del proprietario */

    dev_t     st_rdev;  /* for special files */
                      /* riservato a file speciali */

    off_t    st_size;  /* file size in characters */
                      /* dimensione del file in caratteri */

    time_t    st_atime; /* time last accessed */
                      /* ora dell'ultimo accesso */
```

```

    time_t st_mtime; /* time last modified */
    /* ora dell'ultima modifica */

    time_t st_ctime; /* time inode last changed */
    /* ora dell'ultima modifica dell'inode */
};

}

```

Gran parte di questi valori è spiegata dai commenti. Tipi come `dev_t` e `ino_t` sono definiti nell'intestazione `<sys/types.h>`, che pure deve essere inclusa.

Il membro `st_mode` contiene un insieme di bit che descrivono il file. Le loro definizioni sono incluse in `<sys/stat.h>`; qui ci serve solo la parte che tratta del tipo del file:

```

#define S_IFMT 0160000 /* type of file */
/* tipo di file */

#define S_IFDIR 0040000 /* directory */

#define S_IFCHR 0020000 /* character special */
/* speciale a carattere */

#define S_IFBLK 0060000 /* block special */
/* speciale a blocchi */

#define S_IFREG 0100000 /* regular */
/* regolare */

/* ... */

```

Ora siamo pronti a scrivere il programma `fsize`. Se la modalità ricevuta da `stat` indica che il file non è una directory, la sua dimensione è a portata di mano e quindi può essere visualizzata direttamente. Se il file è una directory, però, occorre trattarla un file per volta, e visto che potrebbe contenere delle directory annidate ( dette “sottodirectory”), il processo è ricorsivo.

La funzione principale si prende cura degli argomenti provenienti dalla riga di comando, passandoli alla funzione `fsize`.

```

#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h> /* flag per lettura e scrittura */
#include <sys/types.h> /* i typedef */
#include <sys/stat.h> /* struttura restituita da stat */
#include "dirent.h"

void fsize(char *);

/* visualizza le dimensioni dei file */
main(int argc, char **argv)
{
    if (argc == 1) /* di default: directory corrente */

```

```

        fsize(".");
    else
        while (--argc > 0)
            fsize(++argv);
    return 0;
}

```

La funzione `fsize` visualizza le dimensioni del file. Se, tuttavia, si tratta di una directory, `fsize` chiama dapprima `dirwalk` per controllare tutti i file al suo interno. Si osservi come i nomi `S_IFMT` e `S_IFDIR` tratti da `<sys/stat.h>` siano impiegati per decidere se il file sia una directory. Le parentesi sono rilevanti, giacché & dà precedenza a ==.

```

int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));

/* fsize: visualizza la dimensione del file "name" */
void fsize(char *name)
{
    struct stat stbuf;

    if (stat(name, &stbuf) == -1)
        fprintf(stderr, "fsize: non riesco ad accedere a %s\n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        dirwalk(name, fsize);
    printf("%ld %s\n", stbuf.st_size, name);
}

```

La funzione `dirwalk` è una procedura di natura generale che applica una funzione a ogni file in una directory. Essa apre la directory, esamina a turno i file presenti con un ciclo, chiamando per ciascuno la funzione, quindi chiude la directory e termina. Dato che `fsize` invoca `dirwalk` per ogni directory, le due funzioni si chiamano l'un l'altra ricorsivamente.

```

#define MAX_PATH 1024

/* dirwalk: applica la funzione fcn a tutti i file di dir */
void dirwalk(char *dir, void (*fcn)(char *))
{
    char name[MAX_PATH];
    Dirent *dp;
    DIR *dfd;

    if ((dfd = opendir(dir)) == NULL) {
        fprintf(stderr, "dirwalk: non riesco ad aprire %s\n", dir);
        return;
    }
    while ((dp = readdir(dfd)) != NULL) {
        if (strcmp(dp->name, ".") == 0
            || strcmp(dp->name, "..") == 0)

```

```

        continue; /* ignora i due elementi corrispondenti alla */
        /* directory stessa e al genitore */
    if (strlen(dir)+strlen(dp->name)+2 > sizeof(name))
        fprintf(stderr, "dirwalk: il nome %s/%s e' troppo lungo\n",
                dir, dp->name);
    else {
        sprintf(name, "%s/%s", dir, dp->name);
        (*fcn)(name);
    }
    closedir(dfd);
}

```

Ogni chiamata a `readdir` restituisce un puntatore a una struttura `Dirent` che riguarda il file successivo, o `NULL` quando non rimane alcun file. Ogni directory contiene sempre dei dati relativi a se stessa e al suo genitore, ossia i due elementi convenzionalmente denominati `"."` e `".."`; questi devono essere evitati, o il programma eseguirà un ciclo perpetuo.

Fino a questo punto, il codice è indipendente dalla rappresentazione delle directory in un dato sistema. Il passo successivo consiste nello stendere versioni minimali di `opendir`, `readdir` e `closedir` per un sistema specifico. Le funzioni che seguono sono per i sistemi di UNIX Version 7 e System V; esse adoperano le informazioni relative alle directory contenute nell'intestazione `<sys/dir.h>`, e cioè:

```

#ifndef DIRSIZ
#define DIRSIZ 14
#endif
struct direct /* directory entry */
/* descrive una directory */
{
    ino_t d_ino; /* inode number */
    /* numero dell'inode */
    char d_name[DIRSIZ]; /* long name does not have '\0' */
    /* un nome lungo non termina con '\0' */
};

```

Alcune versioni del sistema permettono nomi molto più lunghi e hanno strutture più complicate per descrivere le directory.

Il tipo `ino_t` è definito tramite `typedef`, e descrive l'indice di una lista di inode. Sul sistema che usiamo regolarmente è del tipo `unsigned short`, ma questo genere di informazione non va cablata all'interno dei programmi: potrebbe variare per altri sistemi, per cui è preferibile usare `typedef`. Una serie completa di tipi "di sistema" è contenuta in `<sys/types.h>`.

La funzione `opendir` apre la directory, verifica che il file sia una directory (questa volta con la chiamata di sistema `fstat`, che è simile a `stat` ma si applica al descrittore di un file), alloca memoria per una struttura `DIR` e registra le informazioni rilevanti:

```

int fstat(int fd, struct stat *);

/* opendir: apre una directory a beneficio di readdir */

```

```

DIR *opendir(char *dirname)
{
    int fd;
    struct stat stbuf;
    DIR *dp;

    if ((fd = open(dirname, O_RDONLY, 0)) == -1
        || fstat(fd, &stbuf) == -1
        || (stbuf.st_mode & S_IFMT) != S_IFDIR
        || (dp = (DIR *) malloc(sizeof(DIR))) == NULL)
        return NULL;
    dp->fd = fd;
    return dp;
}

```

La funzione `closedir` chiude il file della directory e libera la zona di memoria relativa:

```

/* closedir: chiude una directory aperta da opendir */
void closedir(DIR *dp)
{
    if (dp) {
        close(dp->fd);
        free(dp);
    }
}

```

Infine, `readdir` usa `read` per leggere gli elementi della directory. Se un elemento della directory non è attualmente in uso (perché un file è stato rimosso), il numero dell'inode è zero, e questa posizione viene ignorata. In caso contrario, il numero e il nome dell'inode sono collocati in una struttura statica, restituendo all'utente un puntatore alla struttura stessa. Ogni chiamata sovrascrive le informazioni ottenute da quella precedente.

```

#include <sys/dir.h> /* struttura locale delle directory */

/* readdir: legge in sequenza gli elementi di una directory */
Dirent *readdir(DIR *dp)
{
    struct direct dirbuf; /* struttura locale delle directory */
    static Dirent d; /* valore restituito: struttura portabile */

    while (read(dp->fd, (char *) &dirbuf, sizeof(dirbuf))
          == sizeof(dirbuf)) {
        if (dirbuf.d_ino == 0) /* elemento attualmente non utilizzato */
            continue;
        d.ino = dirbuf.d_ino;
        strncpy(d.name, dirbuf.d_name, DIRSIZ);
        d.name[DIRSIZ] = '\0'; /* assicura la terminazione */
        return &d;
    }
    return NULL;
}

```

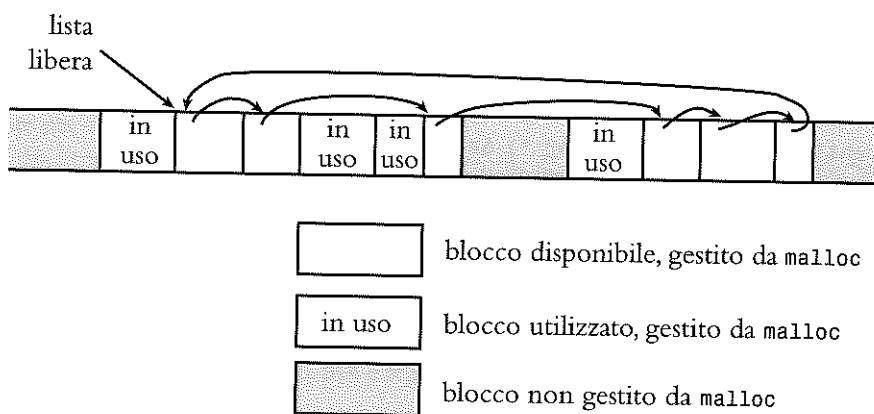
Pur essendo un programma piuttosto specialistico, `fsize` mette in evidenza un paio di idee importanti. In primo luogo, molti programmi non sono “programmi del sistema”, ma sfruttano più semplicemente informazioni proprie del sistema operativo. Per siffatti programmi è vitale che la rappresentazione delle informazioni compaia esclusivamente in certe intestazioni standard, e che il codice includa dette intestazioni anziché cablare le dichiarazioni al suo interno. Secondariamente, si osservi che, procedendo con cura, è possibile creare un’interfaccia agli oggetti dipendenti dal sistema che sia invece relativamente indipendente dal sistema, come ben illustrano le funzioni della libreria standard.

**Esercizio 8.5** Si modifichi il programma `fsize` affinché stampi le altre informazioni contenute nell’inode.

## 8.7 Esempio di allocatore di memoria

Nel Capitolo 5 abbiamo presentato un rudimentale allocatore di memoria basato su una pila (stack). La versione che ora scriviamo non soffre di restrizioni di sorta. Le chiamate a `malloc` e `free` possono avvenire in qualsiasi ordine, e `malloc` chiama il sistema operativo per ottenere la quantità di memoria necessaria. Queste funzioni chiariscono alcune delle problematiche inerenti alla stesura di codice intrinsecamente dipendente dalla macchina di modo che ne risulti per certi versi indipendente, e mostrano anche un’applicazione concreta delle strutture, delle unioni e del costrutto `typedef`.

Invece di estrarre memoria da un vettore di misura predefinita al momento della compilazione, all’occorrenza `malloc` richiederà memoria al sistema operativo. Poiché, in generale, vi sono altri modi di ottenere memoria senza passare per questo allocatore, lo spazio gestito da `malloc` non può essere contiguo. È per questo che lo spazio di memoria gestito da `malloc` è strutturato in una lista di blocchi liberi, ciascuno contenente una dimensione, un puntatore al blocco successivo, e lo spazio di memoria stesso. I blocchi sono disposti in ordine crescente di indirizzo, e l’ultimo blocco (il cui indirizzo è il più grande) punta al primo.



In presenza di una richiesta, la lista libera è scandita fino a che non si trovi un blocco sufficientemente grande. Questo algoritmo è detto del “primo riscontro” (“first fit”), in contrasto con “il miglior riscontro” (“best fit”), che identifica il blocco più piccolo in grado di soddisfare la richiesta. Se il blocco è esattamente della misura desiderata, lo si consegna all’utente, togliendolo dalla lista. Se il blocco è troppo grande, lo si spezza, consegnando la quantità necessaria all’utente e lasciando la parte residua sulla lista. Se non si trova alcun blocco di grandezza sufficiente, tramite il sistema operativo si ottiene un altro segmento di memoria di grandi dimensioni e lo si aggiunge alla lista.

Anche liberare memoria già allocata implica una ricerca nella lista dei blocchi disponibili, al fine di trovare il punto di inserimento appropriato per il blocco da liberare. Se il blocco da rendere nuovamente disponibile si trova a destra o a sinistra di un blocco libero, lo si fonde con questo in un blocco più grande, per non frammentare eccessivamente lo spazio di memoria. Individuare le adiacenze è facile, poiché la lista è mantenuta in ordine crescente di indirizzo.

Un problema, a cui si è accennato nel Capitolo 5, è di garantire che la memoria restituita da `malloc` sia allineata in modo corretto per gli oggetti che vi troveranno posto. Ogni macchina, pur avendo caratteristiche diverse, possiede un singolo tipo massimamente restrittivo: se un oggetto di quel tipo può essere memorizzato a un indirizzo specifico, potranno esserlo anche tutti gli altri oggetti, indipendentemente dal loro tipo. Su alcuni elaboratori il tipo più restrittivo è `double`; su altri, è sufficiente `int` o `long`.

Un blocco libero contiene un puntatore al blocco successivo nella lista, la misura del blocco, e poi lo spazio libero stesso; l’informazione di controllo all’inizio è detta “intestazione” (“header”). Per facilitare l’allineamento, tutti i blocchi sono multipli della dimensione dell’intestazione, che è correttamente allineata. Ciò si ottiene grazie a un’unione contenente la struttura che definisce l’intestazione, assieme a un membro del tipo più restrittivo dal punto di vista dell’allineamento (qui è `long`, a mero titolo d’esempio).

```
typedef long Align; /* per l'allineamento al tipo long */

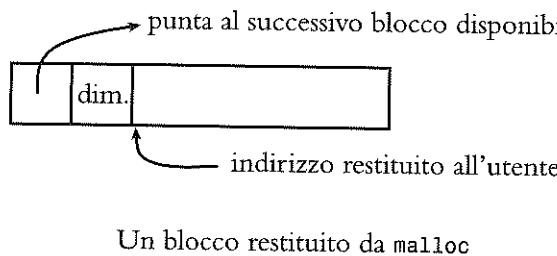
union header { /* intestazione del blocco: */
    struct {
        union header *ptr; /* blocco successivo, se la struttura e' */
        /* sulla lista dei blocchi liberi */ 
        unsigned size; /* dimensione di questo blocco */
    } s;
    Align x; /* forza l'allineamento dei blocchi */
};

typedef union header Header;
```

Il campo `Align` non è mai usato: serve solo a costringere l’intestazione ad allinearsi all’indirizzo che rappresenta il caso peggiore.

In `malloc`, la grandezza richiesta in caratteri è arrotondata al numero appropriato in unità di misura pari alla dimensione di un’intestazione; il blocco allocato contiene un’unità in più, per l’intestazione stessa, e questo è il valore registrato nel campo `size` dell’intestazione.

Il puntatore restituito da `malloc` punta allo spazio disponibile, non all'intestazione medesima. L'utente può disporre a piacimento dello spazio richiesto, ma se si scrive qualcosa al di fuori dello spazio assegnato è probabile che la lista sarà danneggiata.



Il campo dimensione (nella figura definito "dim.") è necessario, perché i blocchi gestiti da `malloc` non sono necessariamente contigui; non è possibile calcolarne le dimensioni con l'aritmetica dei puntatori.

La variabile `base` è usata come punto d'inizio. Se `freep` è `NULL`, com'è alla prima chiamata di `malloc`, si crea una lista degenera dei blocchi liberi, il cui contenuto è un blocco di grandezza zero che punta a se stesso. Si esegue poi comunque la ricerca di un blocco libero di grandezza adeguata nella lista, a partire dal punto in cui l'ultimo blocco è stato trovato (`freep`); questa strategia aiuta a mantenere la lista omogenea. Nell'eventualità che sia trovato un blocco troppo grande, la parte finale di grandezza appropriata è restituita all'utente, in modo che le modifiche necessarie all'intestazione del blocco originale riguardino solo la dimensione. In ogni caso, il puntatore restituito all'utente punta allo spazio libero dentro il blocco, che inizia dall'unità seguente all'intestazione.

```
static Header base;      /* lista vuota per cominciare */
static Header *freep = NULL;    /* punto iniziale della lista */

/* malloc: un allocatore di memoria di portata generale */
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *morecore(unsigned);
    unsigned nunits;

    nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
    if ((prevp = freep) == NULL) { /* la lista non esiste ancora */
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) { /* grande a sufficienza */
            if (p->s.size == nunits) /* la dimensione è esatta */
                prevp->s.ptr = p->s.ptr;
            else { /* alloca la parte finale appropriata */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *) (p+1);
        }
        if (p == freep) /* la lista e' stata esaminata interamente */
            if ((p = morecore(nunits)) == NULL)
                return NULL; /* non c'e' piu' memoria */
    }
}
```

```
p->s.size -= nunits;
p += p->s.size;
p->s.size = nunits;
}
freep = prevp;
return (void *) (p+1);
}
if (p == freep) /* la lista e' stata esaminata interamente */
    if ((p = morecore(nunits)) == NULL)
        return NULL; /* non c'e' piu' memoria */
}
```

La funzione `morecore` ottiene memoria dal sistema operativo. I particolari su come questo avviene variano da un sistema all'altro. Poiché chiedere al sistema ulteriore memoria è un'operazione relativamente dispendiosa, vorremmo evitare che ciò avvenga ogni volta che si invoca `malloc`. È per questo che `morecore` richiede almeno `NALLOC` unità: questo blocco più esteso può essere poi sezionato secondo le esigenze specifiche. Una volta impostato il campo `size`, `morecore` rimette in gioco la memoria in eccesso tramite una chiamata a `free`.

La chiamata di sistema UNIX `sbrk(n)` restituisce un puntatore a `n` byte aggiuntivi di memoria. La funzione `sbrk` restituisce `-1` se non c'è spazio disponibile, benché `NULL` avrebbe rappresentato una soluzione migliore. La costante `-1` deve essere forzata nel tipo `char *` prima di poterla confrontare con il valore restituito da `sbrk`. Come di consueto, l'uso del costrutto `cast` pone la funzione relativamente al riparo dai dettagli relativi alla rappresentazione dei puntatori su macchine differenti. Rimane però l'assunto che puntatori a blocchi diversi restituiti da `sbrk` possano essere confrontati sensatamente: lo standard non garantisce nulla al riguardo, poiché consente il confronto tra puntatori esclusivamente all'interno di uno stesso vettore. Pertanto, la presente versione di `malloc` è portabile solo tra macchine per le quali abbia senso il confronto generale tra puntatori.

```
#define NALLOC 1024 /* numero minimo di unita' da richiedere */

/* morecore: richiede al sistema altra memoria */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1) /* non c'e' proprio piu' spazio */
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *) (up+1));
    return freep;
}
```

Rimane come ultima la funzione `free`. Essa esamina la lista dei blocchi disponibili, partendo da `freep`, in cerca del punto appropriato in cui inserire il blocco libero: ovvero, tra due blocchi esistenti o a un estremo della lista. In entrambi i casi, se il blocco che si sta liberando ha un indirizzo adiacente a quello di uno dei suoi due vicini, si accorpano i blocchi adiacenti in un solo bocco. Bisogna però preoccuparsi che i puntatori indichino gli oggetti giusti e che le misure siano corrette.

```
/* free: inserisce il blocco ap nella lista dei blocchi disponibili */
void free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)ap - 1; /* punta all'intestazione del blocco */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break; /* il blocco liberato e' a un estremo della lista */

    if (bp + bp->s.size == p->s.ptr) { /* fusione con il vicino seguente */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) { /* fusione con il vicino prec. */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

Nonostante l'allocazione di memoria dipenda intrinsecamente dalla macchina, il codice presentato è un esempio di come tale dipendenza possa essere controllata e limitata a una porzione molto ridotta del programma. L'utilizzo di `typedef` e `union` sovrintende all'allineamento (posto che una chiamata di sistema quale `sbrk` fornisca un puntatore appropriato). L'applicazione del costrutto `cast` rende esplicite le conversioni fra puntatori, e rimedia persino a un'interfaccia con il sistema mal congegnata. Malgrado i particolari del codice siano inerenti all'assegnazione di memoria, se ne può adottare l'impostazione generale anche per altre situazioni.<sup>(2)</sup>

2. N.d.R. A proposito del codice di questo paragrafo, l'errata corrige al testo commenta: "Le funzioni `malloc` e `morecore` non sono presentate come parte di un unico programma [...]. "Rendere `morecore` visibile a tutto il file sorgente, anziché alla sola `malloc`, tramite la dichiarazione

`static Header *morecore(unsigned);`  
sarebbe una soluzione migliore dal punto di vista della visibilità e del linkage".

**Esercizio 8.6** La funzione della libreria standard `calloc(n, size)` restituisce un puntatore a `n` oggetti di dimensione `size`, con la memoria inizializzata a zero. Si scriva `calloc`, chiamando il codice `malloc` o modificandolo.

**Esercizio 8.7** La funzione `malloc` accetta di erogare memoria senza verificare la plausibilità della quantità richiesta; `free` ritiene che il blocco che gli si chiede di liberare contenga un campo `size` valido. Il lettore perfeziona queste funzioni mirando a una migliore gestione degli errori.

**Esercizio 8.8** Si scriva una funzione `bfree(p, n)` che liberi un blocco arbitrario `p` di `n` caratteri, inserendolo nella lista dei blocchi disponibili gestita da `malloc` e `free`. Adoperando `bfree`, l'utente può aggiungere in qualsiasi momento alla lista un vettore statico o esterno.

# Manuale di riferimento

## A.1 Introduzione

Questo manuale descrive il linguaggio C specificato nella bozza presentata all'ANSI il 31 Ottobre 1988 perché fosse approvata come "Standard nazionale americano per i sistemi informatici – Linguaggio di programmazione C, X3.159-1989". Il manuale è un'interpretazione dello Standard proposto, non lo standard stesso, ma intende essere una guida affidabile al linguaggio.

Questo documento ricalca in prevalenza lo schema generale dello Standard, che a sua volta segue quello della prima edizione di questo libro, pur avendo nei particolari un'impostazione diversa. A parte la scelta dei nomi di alcune produzioni e l'assenza di una definizione formale dei token del preprocessore, la grammatica fornita per il linguaggio vero e proprio è equivalente a quella dello Standard.

In questo manuale il materiale di commento è scritto in caratteri più piccoli e rientrato, come in questo caso. Molto spesso tali commenti servono a evidenziare i punti in cui il C versione ANSI Standard si discosta dal linguaggio definito nella prima edizione di questo libro, o da migliorie successivamente apportate da vari compilatori.

## A.2 Convenzioni lessicali

Un programma consiste di una o più *unità di traduzione* memorizzate in file. Esso è tradotto in diverse fasi, descritte in §A.12. Le fasi iniziali compiono trasformazioni lessicali di

basso livello, eseguono le direttive introdotte da righe che iniziano con il carattere # ed effettuano la definizione e l'espansione delle macro. Al termine delle operazioni del preprocessore (§A.12), il programma è ridotto a una sequenza di token (cioè, simboli convenzionali).

### A.2.1 Token

Vi sono sei categorie di token: identificatori, parole chiave, costanti, letterali stringa, operatori e altri separatori. Gli “spazi”, ovvero il singolo spazio bianco, la tabulazione verticale e orizzontale, il carattere newline, il salto pagina e i commenti descritti di seguito, vengono ignorati tranne quando usati per separare i token. È necessario usare dello spazio per separare identificatori, parole chiave e costanti che, in mancanza di esso, risulterebbero adiacenti.

Se il flusso in ingresso è stato diviso in token fino a un certo carattere, il token successivo è la più lunga stringa di caratteri che possa costituire un token.

### A.2.2 Commenti

I caratteri /\* introducono un commento, che termina con i caratteri \*/. I commenti annidati, o all'interno di un letterale stringa o carattere, non sono ammessi.

### A.2.3 Identificatori

Un identificatore è una successione di lettere e cifre. Il primo carattere deve essere una lettera, fra le quali si annovera la sottolineatura (\_), cioè il cosiddetto “underscore”. Vi è distinzione fra minuscole e maiuscole. Non c'è limite alla lunghezza degli identificatori, e almeno i primi 31 caratteri di un identificatore interno sono significativi; certe implementazioni possono aumentare questa soglia. Gli identificatori interni comprendono i nomi delle macro del preprocessore e tutti gli altri nomi che non abbiano linkage esterno (§A.11.2). Gli identificatori con linkage esterno sono soggetti a restrizioni più severe: certe implementazioni possono rendere significativi anche solo i primi sei caratteri, senza distinzione fra maiuscole e minuscole.

### A.2.4 Parole chiave

I seguenti identificatori possono essere utilizzati esclusivamente come parole chiave:

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

Alcune implementazioni comprendono anche le parole chiave fortran e asm.

Le parole chiave const, signed e volatile sono una novità dello standard ANSI; enum e void sono state introdotte dopo la prima edizione, ma trovano già vasta applicazione; entry, una parola chiave mai usata nella pratica, è ora stata eliminata.

### A.2.5 Costanti

Esistono diversi generi di costanti, e ciascuna ha un certo tipo; si consulti §A.4.2 per i tipi fondamentali.

*costante:*

- costante-intera*
- costante-carattere*
- costante-virgolamobile*
- costante-enumerazione*

#### A.2.5.1 Costanti intere

Una costante intera costituita da una successione di cifre è interpretata come numero in base otto se comincia per 0 (la cifra zero), e come numero decimale altrimenti. Le costanti in ottale non contengono le cifre 8 e 9. Una successione di cifre preceduta da 0x o 0X (la cifra zero) è interpretata come numero esadecimale. Le cifre esadecimali comprendono le lettere da a o A fino a f o F, i cui valori vanno da 10 a 15.

Una costante intera può essere seguita dalla lettera u o U, a indicare che si tratta di una quantità priva di segno; il suffisso l (lettera elle) o L, invece, denota che trattasi di un intero long.

Il tipo di una costante intera dipende dalla sua forma, dal suo valore e dal suffisso. (Si veda §A.4 per un approfondimento.) Se non ha suffisso ed è decimale, il suo tipo è il primo fra quelli seguenti che permetta di rappresentarne il valore: int, long int, unsigned long int. Se non ha suffisso ed è ottale o esadecimale, ha il primo tipo possibile fra i seguenti: int, unsigned int, long int, unsigned long int. Se reca il suffisso u o U, il tipo è il primo possibile fra: unsigned int, unsigned long int. Se reca il suffisso l o L, il tipo è il primo possibile fra: long int, unsigned long int. Se reca il suffisso UL, il tipo è unsigned long int.

Il trattamento dei tipi delle costanti intere va molto al di là di quanto fatto nella prima edizione, dove si imponeva semplicemente il tipo long alle costanti intere di grandi dimensioni. I suffissi U sono nuovi.

#### A.2.5.2 Costanti di tipo carattere

Una costante (di tipo) carattere è data da una sequenza di uno o più caratteri racchiusi tra apici, come 'x'. Il valore di una costante di un solo carattere è il valore numerico del carattere nel set di caratteri della macchina al momento dell'esecuzione. Il valore di una costante a più caratteri è definito dall'implementazione.

Le costanti carattere non contengono il carattere ' o il carattere newline: al fine di rappresentare questi, e altri caratteri particolari, si possono usare le sequenze di controllo elencate di seguito.

Carattere newline	NL (LF)	\n	Barra inversa	\	\\
Tabulazione orizzontale	HT	\t	Punto interrogativo	?	\?
Tabulazione verticale	VT	\v	Apice	'	\'
Backspace	BS	\b	Virgolette	"	\"
Ritorno del carrello	CR	\r	Numero ottale	ooo	\ooo
Salto pagina	FF	\f	Numero esadecimale	hh	\xhh
Allarme (campanello)	BEL	\a			

La sequenza di controllo \ooo è formata da una barra inversa seguita da 1, 2 o 3 cifre ottali, che si presuppone identifichino il valore del carattere desiderato. Un esempio comune di questa forma è \0 (non seguito da una cifra) che identifica il carattere NUL. La sequenza \xhh è formata da una barra inversa, seguita da x, a cui seguono cifre esadecimali, che si presuppone specifichino il valore del carattere desiderato. Non vi è un limite al numero di cifre, ma il comportamento è indefinito se il valore del carattere che ne risulta supera quello del carattere più grande. Per caratteri di controllo ottali o esadecimali, qualora l'implementazione presupponga che il tipo char sia con segno, il segno si estende al valore come se questo fosse stato convertito in un tipo char tramite cast. Se il carattere successivo a \ non rientra fra quelli specificati, il comportamento è indefinito.

In alcune implementazioni vi è un insieme più ampio di caratteri che il tipo char non può rappresentare. Una costante scritta usando questi caratteri estesi è preceduta da una L, per esempio L'x', e prende il nome di costante carattere estesa (*wide character constant*). Una simile costante ha il tipo wchar\_t, un tipo intero definito nell'intestazione standard <stddef.h>. Come per una costante carattere ordinaria, è ammesso l'uso delle sequenze di controllo ottali o esadecimali; il risultato è indefinito se il valore specificato eccede quello rappresentabile da wchar\_t.

Alcune di queste sequenze di controllo sono nuove, in particolare la rappresentazione esadecimale dei caratteri. Anche i caratteri estesi sono una novità. I set di caratteri di uso comune nelle Americhe e in Europa occidentale possono essere codificati per essere rappresentabili dal tipo char; lo scopo principale di wchar\_t è di includere le lingue asiatiche.

#### A.2.5.3 Costanti con virgola mobile

Una costante con virgola mobile è costituita da una parte intera, un punto decimale, una parte frazionaria, una e o E, un esponente intero con o senza segno e da un eventuale suffisso di tipo, che sia uno fra f, F, l o L. La parte intera e quella frazionaria consistono entrambe di una sequenza di cifre; una delle due (non entrambe) può mancare; il punto decimale o la e con l'esponente (non entrambi) possono mancare. Il tipo è determinato dal suffisso; F o f denotano float, L o l denotano long double; altrimenti il tipo è double.

I suffissi delle costanti con virgola mobile sono nuovi.

#### A.2.5.4 Costanti di enumerazione

Gli identificatori dichiarati come enumeratori (si veda §A.8.4) sono costanti di tipo int.

### A.2.6 Letterali stringa

Un letterale stringa, detto anche costante stringa, è una sequenza di caratteri tra virgolette, come in "....". Una stringa ha come tipo "vettore di caratteri" e come classe di memorizzazione static (si veda §A.4) ed è inizializzata con i caratteri dati. La possibilità di distinguere tra costanti stringa identiche dipende dall'implementazione, e il comportamento di un programma che tenti di modificare un letterale stringa è indefinito.

Letterali stringa adiacenti sono concatenati in una costante singola. Dopo ogni concatenazione, un byte nullo \0 è allegato alla stringa per segnalarne la fine ai programmi che la scandiscano. Le costanti stringa non contengono caratteri newline o virgolette; per rappresentarli sono disponibili le stesse sequenze di controllo delle costanti carattere.

Come per le costanti carattere, i letterali stringa in un insieme esteso di caratteri iniziano con una L, come in L"....". Tali stringhe di caratteri estesi hanno come tipo "vettore di wchar\_t". La concatenazione delle costanti stringa ordinarie con quelle estese è indefinita.

La specifica che le costanti stringa non devono essere distinte, e il divieto di modificarle, sono novità dello standard ANSI, come lo è la concatenazione di costanti stringa adiacenti. Le costanti stringa di caratteri estesi sono nuove.

## A.3 Notazione della sintassi

Nella notazione della sintassi adottata in questo manuale, le categorie sintattiche sono indicate dal carattere *corsivo*, mentre le parole e i caratteri letterali sono in stile *typewriter*. Le categorie alternative sono solitamente elencate in righe separate; in alcuni casi, una lunga serie di brevi alternative viene presentata su un'unica riga, contrassegnata dalla frase "una di". Un simbolo (terminale o no) facoltativo reca il pedice "fac", cosicché, per esempio,

{ *espressione<sub>fac</sub>* }

significa un'espressione facoltativa tra parentesi graffe. La sintassi è riassunta in §A.13.

A differenza della grammatica della prima edizione del libro, quella presentata in questa sede rende esplicativi i diritti di precedenza e l'associatività degli operatori nelle espressioni.

## A.4 Significato degli identificatori

Gli identificatori, o nomi, si riferiscono a diversi elementi: funzioni; etichette; contrassegni delle strutture, delle unioni, delle enumerazioni; membri di strutture o di unioni; costanti di enumerazione; nomi definiti da typedef; oggetti. Un oggetto, chiamato talvolta variabile, è una locazione di memoria, e la sua interpretazione dipende da due attributi: la sua *classe di memorizzazione* e il suo *tipo*. La classe di memorizzazione determina la durata della memoria associata all'oggetto identificato; il tipo determina il significato dei valori trovati nell'oggetto identificato. Un nome ha anche un campo di visibilità (scope), che è la porzione di programma in cui è conosciuto, e un linkage (collegamento), che determina se lo stesso nome in un altro campo di visibilità si riferisce allo stesso oggetto o funzione. Campo di visibilità e linkage sono trattati in §A.11.

#### A.4.1 Classe di memorizzazione

Le classi di memorizzazione sono due, automatica e statica. La classe di memorizzazione di un oggetto è determinata da diverse parole chiave e dal contesto della sua dichiarazione. Gli oggetti automatici sono interni a un blocco (§A.9.3), e vengono eliminati all'uscita dal blocco. Le dichiarazioni nell'ambito di un blocco creano oggetti automatici se non è specificata una classe di memorizzazione, o se si è usato lo specificatore `auto`. Gli oggetti dichiarati `register` sono automatici, e sono (se possibile) memorizzati in registri veloci della macchina.

Gli oggetti statici possono essere interni a un blocco o esterni a tutti i blocchi, ma in ogni caso conservano i propri valori quando il controllo esce e poi rientra nei blocchi e nelle funzioni. All'interno dei blocchi, compresi quelli che forniscono il codice per una funzione, gli oggetti statici sono dichiarati con la parola chiave `static`. Gli oggetti dichiarati al di fuori di tutti i blocchi, allo stesso livello delle definizioni di funzione, sono sempre statici; possono essere resi locali a una specifica unità di traduzione per mezzo della parola chiave `static`, che attribuisce loro un *linkage interno*. Essi diventano accessibili all'intero programma omettendo di specificare la classe di memorizzazione, o per mezzo della parola chiave `extern`, che attribuisce loro un *linkage esterno*.

#### A.4.2 Tipi fondamentali

Vi sono vari tipi fondamentali. L'intestazione standard `<limits.h>` descritta nell'Appendice B definisce i valori più grandi e più piccoli di ogni tipo nell'implementazione locale. I numeri forniti nell'Appendice B rappresentano le grandezze minime possibili.

Gli oggetti dichiarati come caratteri (`char`) hanno grandezza sufficiente a memorizzare qualsiasi membro dell'insieme di caratteri usato nella definizione del linguaggio. Se un carattere autentico di quell'insieme è memorizzato in un oggetto `char`, il suo valore è equivalente al codice intero del carattere, ed è non negativo. È possibile memorizzare altre quantità in variabili `char`, ma la gamma di valori disponibili, e in particolare l'assenza o presenza di segno, dipende dall'implementazione.

I caratteri privi di segno dichiarati `unsigned char` consumano la stessa quantità di spazio dei caratteri normali, ma appaiono sempre non negativi; i caratteri con segno dichiarati `signed char` occupano anch'essi lo stesso spazio dei caratteri normali.

Il tipo `unsigned char` non è presente nella prima edizione di questo libro, ma è di uso comune; `signed char` è nuovo.

Oltre ai tipi `char`, sono disponibili tipi interi di tre grandezze diverse, `short int`, `int` e `long int`. Gli `int` semplici assumono la grandezza naturale suggerita dall'architettura della macchina ospite; le altre misure servono in casi eccezionali. La memoria fornita dagli interi più lunghi è almeno pari a quella dei più brevi, ma l'implementazione può rendere gli interi ordinari equivalenti sia agli interi brevi che a quelli lunghi. I tipi `int` rappresentano tutti valori con segno se non è specificato altrimenti.

Gli interi senza segno, dichiarati tramite la parola chiave `unsigned`, si attengono alle leggi dell'aritmetica modulo  $2^n$ , dove  $n$  è il numero di bit della rappresentazione, e pertanto l'a-

ritmetica con quantità prive di segno non porta mai all'overflow. L'insieme di valori non negativi che possono essere memorizzati in un oggetto con segno è un sottoinsieme dei valori memorizzabili nell'oggetto senza segno corrispondente, e la rappresentazione dei valori coincidenti è la stessa.

I tipi dei numeri con virgola mobile a precisione singola (`float`), doppia (`double`) e multipla (`long double`) possono in realtà coincidere tutti, ma quelli più avanti nella lista devono essere almeno altrettanto precisi dei precedenti.

`long double` è nuovo. La prima edizione rendeva `long float` equivalente a `double`; la locuzione è stata ora soppressa.

Le *enumerazioni* sono tipi unici che hanno valori interi; a ogni enumerazione è connesso un insieme di costanti con nome (§A.8.4). Le enumerazioni si comportano come interi, ma è comune ricevere un avvertimento dal compilatore quando a un oggetto di un particolare tipo enumerativo si assegna qualcosa di diverso dalle sue costanti o da un'espressione del suo tipo.

Gli oggetti appartenenti ai suddetti tipi possono essere interpretati come numeri: saranno pertanto chiamati tipi *aritmetici*. I tipi `char`, gli `int` di qualunque grandezza con o senza segno e anche i tipi enumerativi saranno chiamati collettivamente tipi *integrali* o *interi*. I tipi `float`, `double` e `long double` saranno chiamati tipi *con virgola mobile*.

Il tipo `void` designa un insieme vuoto di valori. È usato come tipo restituito dalle funzioni che non generano alcun valore.

#### A.4.3 Tipi derivati

Ai tipi fondamentali va ad aggiungersi una classe concettualmente infinita di tipi derivati, costruiti a partire dai tipi fondamentali nei modi seguenti:

*vettori* di oggetti di un tipo dato;

*funzioni* che restituiscono oggetti di un tipo dato;

*puntatori* a oggetti di un tipo dato;

*strutture* contenenti una sequenza di oggetti di vari tipi;

*unioni* in grado di contenere uno fra diversi oggetti di vari tipi.

In generale questi metodi di costruzione degli oggetti possono essere applicati ricorsivamente.

#### A.4.4 Qualificatori di tipo

Il tipo di un oggetto può avere qualificatori aggiuntivi. Apporre `const` alla dichiarazione di un oggetto specifica che il suo valore non cambierà; apporre `volatile` specifica che esso possiede particolari proprietà attinenti all'ottimizzazione. Nessuno dei due qualificatori modifica la gamma dei valori o le proprietà aritmetiche dell'oggetto. I qualificatori sono trattati in §A.8.2.

## A.5 Oggetti e lvalue

Un *oggetto* è un'area di memoria contraddistinta da un nome; un *lvalue* è un'espressione che si riferisce a un oggetto. Un esempio ovvio di espressione lvalue è un identificatore dotato di tipo e classe di memorizzazione idonei. Vi sono operatori che producono lvalue: per esempio, se *E* è un'espressione di tipo puntatore, *\*E* sarà un'espressione lvalue che denota l'oggetto puntato da *E*. Il nome “lvalue” (contrazione di *left value*, cioè valore di sinistra) proviene dall'espressione di assegnamento *E1 = E2*, in cui l'operando di sinistra *E1* deve essere un'espressione lvalue. L'analisi dedicata a ogni operatore precisa se esso si aspetta operandi lvalue o se produce un lvalue.

## A.6 Conversioni

Alcuni operatori possono, a seconda dei loro operandi, determinare la conversione del valore di un operando da un tipo a un altro. Questa sezione illustra quale risultato ci si può aspettare da tali conversioni. Il paragrafo A.6.5 riassume le conversioni richieste dagli operatori più frequenti; sarà completato in maniera opportuna dall'analisi di ciascun operatore.

### A.6.1 Promozione integrale

Un carattere, un intero breve o un campo intero di bit, che abbiano il segno o meno, o un oggetto di tipo enumerativo possono essere usati in un'espressione dovunque un intero possa esserlo. Se un *int* può rappresentare tutti i valori del tipo originale, il valore è convertito in *int*; in caso contrario, il valore è convertito in *unsigned int*. Questo processo è chiamato *promozione integrale*.

### A.6.2 Conversioni integrali

Un intero è convertito in un dato tipo senza segno individuando il più piccolo valore non negativo che sia congruo a quell'intero modulo uno in più del valore più grande rappresentabile nel tipo privo di segno. In una rappresentazione in complemento a due, ciò equivale a troncare a sinistra se la successione di bit del tipo privo di segno è più corta, e al riempimento con zeri dei valori privi di segno, insieme all'estensione del segno per i valori con segno, se il tipo privo di segno è più lungo.

Quando un intero è convertito in un tipo con segno, il valore resta immutato se è rappresentabile nel nuovo tipo; altrimenti, il valore dipende dall'implementazione.

### A.6.3 Interi e numeri con virgola mobile

Quando un valore con virgola mobile è convertito in intero, la parte frazionaria viene scartata; se il valore che ne risulta non può essere rappresentato nel tipo intero, il comportamento è indefinito. In particolare, ciò che risulta dalla conversione di valori con virgola mobile negativi in tipi integrali privi di segno non è specificato.

Quando un valore di tipo integrale è convertito in virgola mobile, e il valore rientra nella gamma rappresentabile ma non è rappresentabile con esattezza, il risultato può essere il successivo valore rappresentabile, o il precedente. Se il risultato esula dalla gamma rappresentabile, il comportamento è indefinito.

### A.6.4 Tipi con virgola mobile

Quando un valore con virgola mobile è convertito in un tipo con virgola mobile di precisione uguale o maggiore, il valore è immutato. Quando un valore con virgola mobile più preciso è convertito in un tipo con virgola mobile meno preciso, e il valore rientra nella gamma rappresentabile, il risultato può essere approssimato al valore appena più grande o a quello appena più piccolo. Se il risultato non rientra nella gamma, il comportamento è indefinito.

### A.6.5 Conversioni aritmetiche

Molti operatori causano conversioni e producono tipi di risultati in modo simile. L'effetto è di ridurre gli operandi a un tipo comune, che è anche il tipo del risultato. Questo schema prende il nome convenzionale di *conversioni aritmetiche abituali*.

Prima di tutto, se uno degli operandi è *long double*, l'altro è convertito in *long double*.

Altrimenti, se uno degli operandi è *double*, l'altro è convertito in *double*.

Altrimenti, se uno degli operandi è *float*, anche l'altro è convertito in *float*.

Altrimenti, le promozioni integrali sono applicate a entrambi gli operandi; quindi, se uno degli operandi è *unsigned long int*, l'altro è convertito in *unsigned long int*.

Altrimenti, se un operando è *long int* e l'altro è *unsigned int*, il risultato dipende dalla capacità di *long int* di rappresentare tutti i valori *unsigned int*; se è così, l'operando *unsigned int* è convertito in *long int*; altrimenti, sono entrambi convertiti in *unsigned long int*.

Altrimenti, se un operando è *long int*, l'altro è convertito in *long int*.

Altrimenti, se uno degli operandi è *unsigned int*, anche l'altro è convertito in *unsigned int*.

Altrimenti, entrambi gli operandi hanno tipo *int*.

Si noti che sono intervenute due modifiche. In primo luogo, l'aritmetica sugli operandi *float* può compiersi in precisione singola, anziché doppia; nella prima edizione tutta l'aritmetica con virgola mobile era in doppia precisione. Inoltre, i tipi più brevi senza segno, quando interagiscono con un tipo più lungo con segno, non estendono la proprietà d'essere privi di segno al tipo del risultato: nella prima edizione, il tipo privo di segno prelevava sempre. Le nuove regole sono lievemente più complicate, ma riducono un po' le sorprese che possono scaturire dall'incontro tra quantità con e senza segno. Esiti imprevedibili possono ancora accadere quando un'espressione senza segno è messa a confronto con un'espressione con segno della medesima grandezza.

### A.6.6 Puntatori e interi

Un'espressione di tipo integrale può essere aggiunta a un puntatore o sottratta da esso; in tale ipotesi l'espressione integrale è convertita secondo quanto precisato nell'analisi dell'operatore di addizione (§A.7.7).

Due puntatori a oggetti dello stesso tipo, nello stesso vettore, possono essere sottratti; il risultato è convertito in un intero come precisato sempre in §A.7.7.

Un'espressione integrale costante di valore 0, o la stessa espressione convertita tramite cast al tipo `void *`, può essere convertita, per mezzo di un cast o di un assegnamento, o per confronto, in un puntatore di qualunque tipo. Il puntatore nullo risultante è uguale a un altro puntatore nullo dello stesso tipo, ma diverso da qualsiasi puntatore a una funzione o a un oggetto.

Sono possibili altre conversioni con i puntatori, ma presentano aspetti legati all'implementazione, e devono essere specificate da un operatore esplicito per la conversione del tipo, o da un cast (§§A.7.5 e A.8.8).

Un puntatore può essere convertito in un tipo integrale di grandezza sufficiente a contenere; la misura necessaria dipende dall'implementazione. La corrispondenza specifica dipende anch'essa dall'implementazione.

Un oggetto di tipo integrale può essere esplicitamente convertito in un puntatore. La corrispondenza riporta sempre un intero sufficientemente grande al puntatore da cui, in origine, era stato convertito, ma per tutto il resto è dipendente dall'implementazione.

Un puntatore a un tipo può essere convertito in un puntatore a un tipo diverso. Il puntatore che ne risulta può provocare eccezioni di indirizzamento se l'oggetto a cui si riferisce non è opportunamente allineato in memoria. È garantito che un puntatore a un oggetto possa essere convertito in un puntatore a un oggetto il cui tipo abbia vincoli di allineamento non più restrittivi di quelli dell'oggetto iniziale, in modo che la riconversione del puntatore risultante al suo tipo originario non produca modifiche di sorta rispetto alla situazione di partenza. La nozione di "allineamento" è legata all'implementazione, ma oggetti di tipo `char` hanno requisiti di allineamento minimi. Come descritto in §A.6.8, un puntatore può anche essere prima convertito nel tipo `void *` e poi di nuovo nel tipo originario senza modifiche.

Un puntatore può essere convertito in un altro puntatore il cui tipo è identico a eccezione dell'aggiunta o dell'eliminazione di qualificatori (§§A.4.4, A.8.2) del tipo di oggetto a cui il puntatore si riferisce. Se vengono aggiunti qualificatori, il nuovo puntatore è equivalente al vecchio tranne per le restrizioni implicate dai nuovi qualificatori. Se si cancellano qualificatori, le operazioni sull'oggetto sottostante restano vincolate ai qualificatori espressi nella sua dichiarazione.

Infine, un puntatore a una funzione può essere convertito in un puntatore a una funzione di tipo diverso. Invocare la funzione indicata dal puntatore convertito produce risultati dipendenti dall'implementazione; peraltro, se il puntatore convertito viene riconvertito nel suo tipo originale, il risultato è identico al puntatore iniziale.

### A.6.7 Void

Il valore (inesistente) di un oggetto `void` non può essere usato in alcun modo, né può essere applicata la conversione, implicita o esplicita, a qualsivoglia tipo diverso da `void`. Un'espressione `void`, denotando un valore inesistente, può essere utilizzata solo dove non è richiesto un valore, per esempio come un'istruzione espressione (§A.9.2) o come operando sinistro dell'operatore virgola (§A.7.18).

Un'espressione può essere convertita nel tipo `void` per mezzo di un cast. Un tale cast, per esempio, testimonia che il valore di una chiamata a una funzione impiegata come espressione istruzione non è utilizzato dal programma.

`void` non appariva nella prima edizione di questo libro, ma successivamente è diventato comune.

### A.6.8 Puntatori a void

Ogni puntatore a un oggetto può essere convertito nel tipo `void *` senza perdita di informazioni. Qualora il risultato sia nuovamente convertito nel tipo originale del puntatore, coinciderà con il puntatore iniziale. A differenza delle conversioni da puntatore a puntatore esaminate in §A.6.6, che in genere esigono un cast esplicito, i puntatori possono essere assegnati a puntatori del tipo `void *` e viceversa, e possono essere confrontati con essi.

Questa interpretazione dei puntatori `void *` è nuova; in precedenza, i puntatori `char *` ricoprivano il ruolo di puntatore generico. Lo standard ANSI approva in particolare l'incontro di puntatori `void *` con puntatori a oggetti negli assegnamenti e nelle espressioni relazionali, mentre richiede l'uso esplicito di cast per altre combinazioni di puntatori.

## A.7 Espressioni

L'ordine prescelto dei sottoparagrafi che seguono rispetta il diritto di precedenza fra operatori delle espressioni. Quindi, per esempio, le espressioni indicate come gli operandi di `+` (§A.7.7) sono quelle definite nei sottoparagrafi precedenti (§§A.7.1-A.7.6). All'interno di ciascun sottoparagrafo, gli operatori hanno la stessa precedenza. L'associatività da sinistra o da destra è specificata all'interno del relativo sottoparagrafo. Alla grammatica in §A.13 sono annessi i diritti di precedenza e l'associatività degli operatori.

Precedenza e associatività degli operatori sono specificate minuziosamente, ma l'ordine di valutazione delle espressioni è, con alcune eccezioni, indefinito, anche se le sottoespressioni comportano effetti collaterali. Quindi, a meno che la definizione di un operatore non assicuri che i propri operandi saranno valutati in un certo ordine, l'implementazione è libera di valutare gli operandi in qualsiasi ordine, o persino di interfogliarne la valutazione. Tuttavia, ciascun operatore combina i valori ottenuti dai propri operandi in maniera compatibile con l'albero sintattico dell'espressione in cui appare.

Questa regola revoca la precedente discrezionalità nel riordinare le espressioni con operatori matematicamente commutativi e associativi, ma non necessariamente computazionalmente associativi. La modifica interessa solo l'accuratezza dei calcoli decimali, e le situazioni che possono indurre overflow.

La gestione di overflow, i controlli sulla divisione e altre eccezioni nella valutazione delle espressioni non sono definiti dal linguaggio. Molte implementazioni esistenti del C ignorano l'overflow nella valutazione delle espressioni integrali con segno e degli assegnamenti, ma questa condotta non è richiesta dallo standard. Il trattamento della divisione per 0, e di tutte le eccezioni inerenti i numeri con virgola mobile, è difforme nelle varie implementazioni; talvolta si può modificare la gestione di tali eccezioni tramite una funzione di libreria non standard.

### A.7.1 Generazione di puntatori

Se il tipo di un'espressione o sottoespressione è “vettore di  $T$ ”, per un certo tipo  $T$ , il valore dell'espressione è un puntatore al primo oggetto del vettore, e il tipo dell'espressione è mutato in “puntatore a  $T$ ”. Questa conversione non avviene se l'espressione è l'operando dell'operatore unario &, o di ++, --, sizeof, o come operando di sinistra di un operatore di assegnamento o dell'operatore “.”. Analogamente, un'espressione del tipo “funzione che restituisce  $T$ ”, salvo quando sia impiegata come operando dell'operatore &, viene convertita in “puntatore alla funzione che restituisce  $T$ ”.

### A.7.2 Espressioni primarie

Sono espressioni primarie gli identificatori, le costanti, le stringhe o le espressioni tra parentesi.

*espressione-primaria:*  
*identificatore*  
*costante*  
*stringa*  
 $(\text{espressione})$

Un identificatore è un'espressione primaria, posto che sia stata opportunamente dichiarato come analizzato di seguito. Il suo tipo è precisato dalla dichiarazione. Un identificatore è un lvalue se si riferisce a un oggetto (§A.5) e se il suo tipo è aritmetico, struttura, unione o puntatore.

Una costante è un'espressione primaria. Il suo tipo dipende dalla sua forma, come descritto in §A.2.5.

Un letterale stringa è un'espressione primaria. Originariamente ha come tipo “vettore di char” (per stringhe di caratteri estesi, “vettore di wchar\_t”), ma in conformità alla regola introdotta in §A.7.1, esso è abitualmente modificato in “puntatore a char” (wchar\_t): ciò che ne risulta è un puntatore al primo carattere della stringa. La conversione non si applica a determinati inizializzatori; si veda §A.8.7.

Un'espressione tra parentesi è un'espressione primaria di tipo e valore identici a quelli che avrebbe l'espressione senza parentesi. La presenza delle parentesi non modifica l'eventuale natura lvalue dell'espressione.

### A.7.3 Espressioni postfisse

Gli operatori nelle espressioni postfisse si raggruppano da sinistra a destra.

*espressione-postfissa:*  
*espressione-primaria*  
*espressione-postfissa [ espressione ]*  
*espressione-postfissa ( lista-argomenti-espressione<sub>fac</sub> )*  
*espressione-postfissa . identificatore*  
*espressione-postfissa -> identificatore*  
*espressione-postfissa ++*  
*espressione-postfissa --*  
*lista-argomenti-espressione:*  
*espressione-assegname*  
*lista-argomenti-espressione , espressione-assegname*

#### A.7.3.1 Riferimenti ai vettori

Un'espressione postfissa seguita da un'espressione tra parentesi quadre è un'espressione postfissa che denota un riferimento tramite indice a un elemento di un vettore. Una delle due espressioni deve avere come tipo “puntatore a  $T$ ”, dove  $T$  è un certo tipo, mentre l'altra deve avere tipo integrale; il tipo dell'espressione indice è  $T$ . L'espressione  $E1[E2]$  è identica (per definizione) a  $*((E1)+(E2))$ . Si veda §A.8.6.2 per ulteriori approfondimenti.

#### A.7.3.2 Chiamate a funzioni

Una chiamata a una funzione è un'espressione postfissa, detta designatore della funzione, seguita da parentesi che racchiudono una lista vuota oppure contenente una serie di espressioni di assegnamento separate da una virgola (§A.7.17) che rappresentano gli argomenti della funzione. Se l'espressione postfissa consiste in un identificatore che non è stato dichiarato nel campo di visibilità corrente, l'identificatore è implicitamente dichiarato come se la dichiarazione

```
extern int identificatore();
```

fosse stata inserita nel blocco più interno contenente la chiamata alla funzione. L'espressione postfissa (dopo l'eventuale dichiarazione implicita e l'eventuale generazione del puntatore, §A.7.1) deve essere di tipo “puntatore a una funzione che restituisce  $T$ ”, per un tipo  $T$ , e il valore della chiamata alla funzione ha tipo  $T$ .

Nella prima edizione il tipo era limitato a “funzione”, e un esplicito operatore \* era necessario per invocare funzioni tramite puntatori. Lo standard ANSI avalla la pratica di alcuni compilatori esistenti, perché consente di usare la stessa sintassi per le chiamate a funzioni e a funzioni specificate da puntatori. La vecchia sintassi si può ancora usare.

Il termine *argomento* si applica a un'espressione passata da una chiamata alla funzione; il termine *parametro* è usato per un oggetto in ingresso (o per il suo identificatore) ricevuto

dalla definizione di una funzione, o descritto nella dichiarazione di una funzione. Le rispettive diciture “argomento (parametro) attuale” e “argomento (parametro) formale” sono talvolta usate per la medesima distinzione.

Preliminarmente alla chiamata a una funzione, viene eseguita una copia di ogni argomento; il passaggio di tutti gli argomenti avviene rigorosamente per valore. Una funzione può cambiare i valori dei propri oggetti-parametro, che sono copie delle espressioni degli argomenti, ma queste modifiche non hanno effetto sui valori degli argomenti. Tuttavia, è possibile passare un puntatore, accettando la possibilità che la funzione invocata modifichi il valore dell’oggetto da esso puntato.

Le funzioni possono essere dichiarate secondo due modalità. Con la più recente, i tipi dei parametri sono esplicativi e sono parte del tipo della funzione; tale dichiarazione è anche chiamata prototipo della funzione. Con la vecchia modalità, i tipi dei parametri non sono specificati. Le dichiarazioni delle funzioni sono analizzate in §§A.8.6.3 e A.10.1.

Se la dichiarazione di una funzione visibile dal punto in cui avviene una chiamata segue la vecchia modalità, si applica la promozione di default a ogni argomento, come segue: la promozione integrale (§A.6.1) è applicata a ciascun argomento integrale, e ogni argomento `float` è convertito in `double`. L’effetto della chiamata è indefinito se il numero di argomenti non concorda con il numero di parametri nella definizione della funzione, o se il tipo di un argomento dopo la promozione non concorda con quello del parametro corrispondente. La concordanza richiesta fra tipi dipende dalla modalità, nuova o vecchia, della definizione della funzione. Se è vecchia, il confronto è tra il tipo promosso dell’argomento della chiamata e il tipo promosso del parametro; se è nuova, il tipo promosso dell’argomento deve essere lo stesso del parametro, senza promozione.

Se la dichiarazione della funzione nell’ambito di una chiamata adotta la nuova modalità, gli argomenti sono convertiti, come per assegnamento, nei tipi dei parametri corrispondenti del prototipo della funzione. Il numero di argomenti deve corrispondere a quello dei parametri esplicitamente descritti, a meno che la lista dei parametri della dichiarazione termini con l’ellissi (`, ...`). In tal caso, il numero di argomenti deve egualizzare o superare il numero dei parametri; gli argomenti successivi ai parametri con tipo esplicito incorrono nella promozione di default, come descritto precedentemente. Se la definizione della funzione segue la vecchia modalità, il tipo di ciascun parametro nel prototipo visibile alla chiamata deve concordare con il parametro che gli corrisponde nella definizione, dopo che il tipo del parametro per la definizione è stato sottoposto alla promozione.

Queste regole sono particolarmente complicate perché devono conciliare prassi vecchie e prassi nuove. È meglio evitare, se possibile, di adottare la vecchia e la nuova modalità in modo promiscuo.

L’ordine di valutazione degli argomenti è indeterminato; si osservi che esistono reali differenze tra i vari compilatori. Tuttavia, gli argomenti e il designatore della funzione devono essere valutati nella loro interezza, effetti collaterali inclusi, prima di accedere alla funzione. Sono permesse chiamate ricorsive a qualunque funzione.

### A.7.3.3 Riferimenti alle strutture

Un’espressione seguita da un punto e da un identificatore è un’espressione postfissa. La prima espressione, ossia l’operando, deve essere una struttura o un’unione, e l’identificatore deve designare un membro della struttura o dell’unione. Il valore è il membro della struttura o dell’unione che è stato designato, mentre il suo tipo è il tipo del membro. L’espressione è un lvalue se la prima espressione è un lvalue e se il tipo della seconda espressione non è un vettore.

Un’espressione postfissa seguita da una freccia (che si ottiene con `->`) seguita da un identificatore è un’espressione postfissa. La prima espressione, ossia l’operando, deve essere un puntatore a una struttura o a un’unione, e l’identificatore deve denotare un membro della struttura o dell’unione. Il risultato si riferisce al membro denotato della struttura o dell’unione a cui l’espressione puntatore punta, mentre il tipo è il tipo del membro; il risultato è un lvalue se il tipo non è un vettore.

Di conseguenza l’espressione `E1->MOS` è identica a `(*E1) .MOS`. Le strutture e le unioni sono trattate in §A.8.3.

Nella prima edizione di questo libro, era già vigente la regola che il nome di un membro in un’espressione di tale genere dovesse appartenere alla struttura o all’unione citata nell’espressione postfissa; peraltro, in una nota si ammetteva che la regola non aveva applicazione rigorosa. I compilatori recenti, e l’ANSI, la applicano puntualmente.

### A.7.3.4 Incremento postfisso

Un’espressione postfissa seguita da un operatore `++` o `--` è un’espressione postfissa. Il valore dell’espressione è il valore dell’operando. Una volta riconosciuto il valore, l’operando viene aumentato (`++`) o diminuito (`--`) di 1. L’operando deve essere un lvalue; si veda l’analisi relativa agli operatori additivi (§A.7.7) e all’assegnamento (§A.7.17) per ulteriori limitazioni all’operando e altri dettagli relativi all’operazione. Il risultato non è un lvalue.

## A.7.4 Operatori unari

Le espressioni con operatori unari si raggruppano da destra a sinistra.

*espressione-unaria:*

- espressione-postfissa*
- ++ espressione-unaria*
- espressione-unaria*
- operatore-unario espressione-cast*
- sizeof espressione-unaria*
- sizeof ( nome-tipo )*

*operatore-unario:* uno fra

- `&`
- `*`
- `+`
- `-`
- `~`
- `!`

#### A.7.4.1 Operatori di incremento prefisso

Un'espressione unaria preceduta da un operatore `++` o `--` è un'espressione unaria. L'operando è aumentato (`++`) o diminuito (`--`) di 1. Il valore dell'espressione è il valore dopo l'incremento (o il decremento). L'operando deve essere un lvalue; si veda l'analisi sugli operatori additivi (§A.7.7) e sull'assegnamento (§A.7.17) per ulteriori limitazioni all'operando e altri dettagli relativi all'operazione. Il risultato non è un lvalue.

#### A.7.4.2 Operatore di indirizzamento

L'operatore unario `&` prende l'indirizzo del proprio operando. L'operando deve essere un lvalue che non si riferisce a un campo di bit o a un oggetto dichiarato come `register`, oppure deve avere il tipo di una funzione. Il risultato è un puntatore all'oggetto o alla funzione a cui lo lvalue si riferisce. Se il tipo dell'operando è `T`, il tipo del risultato è “puntatore a `T`”.

#### A.7.4.3 Operatore di indirezione

L'operatore unario `*` denota l'indirezione, e restituisce l'oggetto o la funzione a cui punta il proprio operando. È un lvalue se l'operando è un puntatore a un oggetto che abbia tipo aritmetico, struttura, unione o puntatore. Se il tipo dell'espressione è “puntatore a `T`”, il tipo del risultato è `T`.

#### A.7.4.4 Operatore unario `+`

L'operando dell'operatore unario `+` deve avere tipo aritmetico, e il risultato è dato dal valore dell'operando. A un operando integrale si applica la promozione integrale. Il tipo del risultato è il tipo dell'operando promosso.

Il `+` unario è una novità dello standard ANSI. È stato aggiunto per simmetria con il `-` unario.

#### A.7.4.5 Operatore unario `-`

L'operando dell'operatore unario `-` deve avere tipo aritmetico, e il risultato è il negativo del proprio operando. Un operando integrale subisce la promozione integrale. Il negativo di una quantità senza segno è calcolato sottraendo il valore promosso dal valore più grande del tipo promosso e aggiungendo uno; ma il negativo di zero è zero. Il tipo del risultato è quello dell'operando promosso.

#### A.7.4.6 Operatore di complemento a uno

L'operando dell'operatore `-` deve avere tipo integrale, e il risultato è il complemento a uno del proprio operando. Si applicano le promozioni integrali. Se l'operando è privo di segno, il risultato si calcola sottraendo il valore dal valore più grande del tipo promosso. Se l'operando ha il segno, il risultato è calcolato convertendo l'operando promosso nel tipo corrispondente privo di segno, applicando `-` e riconvertendo il risultato nel tipo con segno. Il tipo del risultato è quello dell'operando promosso.

#### A.7.4.7 Operatore di negazione logica

L'operando dell'operatore `!` deve avere tipo aritmetico o essere un puntatore; il risultato è 1 se il confronto del valore del suo operando con zero è positivo, altrimenti è zero. Il tipo del risultato è `int`.

#### A.7.4.8 Operatore `sizeof`

Il risultato dell'operatore `sizeof` è il numero di byte necessari a memorizzare un oggetto del tipo del proprio operando. L'operando è un'espressione, che non viene valutata, oppure un nome di tipo tra parentesi. Quando `sizeof` è applicato a un `char`, il risultato è 1; se è applicato a un vettore, il risultato è il numero totale di byte del vettore. Quando è applicato a una struttura o a un'unione, il risultato è il numero di byte nell'oggetto, inclusi gli eventuali riempimenti per far sì che l'oggetto sia allineato a un vettore: la grandezza di un vettore di `n` elementi è data da `n` volte la grandezza di un elemento. L'operatore non può essere applicato a un operando del tipo di una funzione, o di tipo incompleto, o a un campo di bit. Il risultato è una costante integrale priva di segno; il tipo specifico è definito dall'implementazione. L'intestazione standard `<stddef.h>` definisce questo tipo come `size_t`.

### A.7.5 Cast

Un'espressione unaria preceduta dal nome di un tipo tra parentesi provoca la conversione del valore dell'espressione nel tipo in questione.

*espressione-cast:*  
*espressione-unaria*  
 $(\text{nome-tipo}) \text{ espressione-cast}$

Questo costrutto è detto *cast*. I nomi dei tipi sono descritti in §A.8.8. I risultati delle conversioni sono descritti in §A.6. Un'espressione con un cast non è un lvalue.

### A.7.6 Operatori moltiplicativi

Gli operatori moltiplicativi `*`, `/` e `%` si raggruppano da sinistra a destra.

*espressione-moltiplicativa:*  
*espressione-cast*  
*espressione-moltiplicativa \* espressione-cast*  
*espressione-moltiplicativa / espressione-cast*  
*espressione-moltiplicativa % espressione-cast*

Gli operandi di `*` e `/` devono avere tipo aritmetico; gli operandi di `%` devono avere tipo integrale. Le conversioni aritmetiche abituali sono effettuate sugli operandi, e preannunciano il tipo del risultato.

L'operatore binario `*` denota moltiplicazione.

L'operatore binario `/` fornisce il quoziente, e l'operatore `%` il resto, della divisione del primo operando per il secondo; se il secondo operando è 0, il risultato è indefinito. Altrimenti,

è sempre vero che  $(a/b)*b + a \% b$  è uguale ad  $a$ . Se entrambi gli operandi sono non negativi, il resto è non negativo e più piccolo del divisore; in caso contrario, è garantito solo che il valore assoluto del resto sia più piccolo del valore assoluto del divisore.

### A.7.7 Operatori additivi

Gli operatori additivi  $+$  e  $-$  si raggruppano da sinistra a destra. Se gli operandi hanno tipo aritmetico, sono effettuate le conversioni aritmetiche abituali. Vi sono alcune possibilità aggiuntive, in merito al tipo, per ogni operatore.

*espressione-additiva:*

*espressione-moltiplicativa*

*espressione-additiva*  $+$  *espressione-moltiplicativa*

*espressione-additiva*  $-$  *espressione-moltiplicativa*

Il risultato dell'operatore  $+$  è la somma degli operandi. Un puntatore a un oggetto di un vettore e un valore di qualsiasi tipo integrale possono essere sommati. Il secondo viene convertito in uno scostamento dall'inizio del vettore moltiplicandolo per la grandezza dell'oggetto puntato dal puntatore. La somma è un puntatore di tipo uguale al puntatore originale, e punta a un altro oggetto nello stesso vettore, opportunamente scostato dall'oggetto originale. Pertanto se  $P$  è un puntatore a un oggetto di un vettore, l'espressione  $P+1$  è un puntatore all'oggetto successivo nel vettore. Se il puntatore derivante dalla somma punta al di fuori dei limiti del vettore, eccetto per la prima posizione dopo il limite superiore, il risultato è indefinito.

La regola che permette ai puntatori di superare il confine di un vettore è nuova. Ciò legittima un idioma diffuso per la scansione degli elementi di un vettore.

Il risultato dell'operatore  $-$  è la differenza tra gli operandi. Si può sottrarre a un puntatore un valore di qualunque tipo integrale, e valgono le medesime conversioni e condizioni applicate all'addizione.

Se si sottraggono due puntatori a oggetti dello stesso tipo, il risultato è un valore integrale con segno che rappresenta la distanza tra gli oggetti puntati; i puntatori a oggetti successivi differiscono di 1. Il tipo del risultato dipende dall'implementazione, ma è definito come `ptrdiff_t` nell'intestazione standard `<stddef.h>`. Il valore rimane indefinito salvo che i puntatori puntino a oggetti dello stesso vettore; tuttavia se  $P$  punta all'ultimo membro di un vettore,  $(P+1)-P$  ha valore 1.

### A.7.8 Operatori di scorrimento

Gli operatori di scorrimento  $<<$  e  $>>$  si raggruppano da sinistra a destra. Entrambi prevedono operandi integrali, che sono soggetti a promozioni integrali. Il tipo del risultato è quello dell'operando sinistro promosso. Il risultato non è definito se l'operando di destra è negativo, ovvero maggiore di o uguale al numero di bit nel tipo dell'espressione di sinistra.

*espressione-scorrimento:*

*espressione-additiva*

*espressione-scorrimento*  $<<$  *espressione-additiva*

*espressione-scorrimento*  $>>$  *espressione-additiva*

Il valore di  $E1<<E2$  è quello di  $E1$  (da interpretare come una sequenza di bit) cui si applica uno scorrimento a sinistra di  $E2$  bit; in mancanza di overflow, questo ammonta alla moltiplicazione per  $2^{E2}$ . Il valore di  $E1>>E2$  è  $E1$  cui si applica uno scorrimento a destra di  $E2$  posizioni di bit. Lo scorrimento a destra equivale alla divisione per  $2^{E2}$  se  $E1$  è privo di segno o ha un valore non negativo; in caso diverso il risultato è definito dall'implementazione.

### A.7.9 Operatori relazionali

Gli operatori relazionali si raggruppano da sinistra a destra, ma questo fatto non è molto utile;  $a < b < c$  è analizzato sintatticamente come  $(a < b) < c$ , e  $a < b$  dà per risultato 0 o 1.

*espressione-relazionale:*

*espressione-scorrimento*

*espressione-relazionale*  $<$  *espressione-scorrimento*

*espressione-relazionale*  $>$  *espressione-scorrimento*

*espressione-relazionale*  $\leq$  *espressione-scorrimento*

*espressione-relazionale*  $\geq$  *espressione-scorrimento*

Gli operatori  $<$  (minore),  $>$  (maggiore),  $\leq$  (minore di o uguale a) e  $\geq$  (maggiore di o uguale a) generano tutti 0 se la relazione specificata è falsa e 1 se è vera. Il tipo del risultato è `int`. Le conversioni aritmetiche abituali si applicano agli operandi aritmetici. I puntatori a oggetti del medesimo tipo (senza tener conto dei qualificatori) possono essere confrontati; il risultato dipende dalle relative posizioni nello spazio degli indirizzi degli oggetti puntati. Il confronto tra puntatori è definito esclusivamente per parti dello stesso oggetto: se due puntatori puntano allo stesso oggetto semplice, risultano uguali; se si tratta di puntatori a membri della stessa struttura, i puntatori a oggetti dichiarati più avanti nella struttura risultano maggiori; se sono puntatori a membri della stessa unione, risultano uguali; qualora i puntatori si riferiscano ai membri di un vettore, il loro confronto è equivalente a quello tra gli indici corrispondenti. Se  $P$  punta all'ultimo membro di un vettore,  $P+1$  risulta maggiore di  $P$ , anche se  $P+1$  punta al di fuori del vettore. In tutti gli altri casi, il confronto tra puntatori è indefinito.

Queste regole attenuano i vincoli stabiliti nella prima edizione, permettendo il confronto tra puntatori a membri diversi di una struttura o unione. Inoltre legittimano il confronto con un puntatore appena oltre il confine di un vettore.

### A.7.10 Operatori di uguaglianza

*espressione-uguaglianza:*

*espressione-relazionale*

*espressione-uguaglianza*  $=$  *espressione-relazionale*

*espressione-uguaglianza*  $\neq$  *espressione-relazionale*

Gli operatori  $=$  (uguale a) e  $\neq$  (non uguale a) sono analoghi agli operatori relazionali tranne per il fatto che hanno precedenza minore. (Quindi  $a < b == c < d$  è 1 ogni volta che  $a < b$  e  $c < d$  abbiano lo stesso valore di verità.)

Gli operatori di uguaglianza rispettano le stesse regole degli operatori relazionali, ma consentono possibilità maggiori: un puntatore può essere confrontato con un'espressione integrale costante di valore 0, o con un puntatore a void. Si veda §A.6.6.

### A.7.11 Operatore AND bit per bit

*espressione-AND:*

espressione-uguaglianza

espressione-AND & espressione-uguaglianza

Si effettuano le conversioni aritmetiche abituali; il risultato è la funzione AND bit per bit degli operandi. L'operatore è applicato solo agli operandi integrali.

### A.7.12 Operatore OR bit per bit esclusivo

*espressione-OR-esclusivo:*

espressione-AND

espressione-OR-esclusivo ^ espressione-AND

Si effettuano le conversioni aritmetiche abituali; il risultato è la funzione OR bit per bit esclusivo degli operandi. L'operatore è applicato solo agli operandi integrali.

### A.7.13 Operatore OR bit per bit inclusivo

*espressione-OR-inclusivo:*

espressione-OR-esclusivo

espressione-OR-inclusivo | espressione-OR-esclusivo

Si effettuano le conversioni aritmetiche abituali; il risultato è la funzione OR bit per bit inclusivo degli operandi. L'operatore è applicato solo agli operandi integrali.

### A.7.14 Operatore AND logico

*espressione-AND-logico:*

espressione-OR-inclusivo

espressione-AND-logico && espressione-OR-inclusivo

L'operatore **&&** si raggruppa da sinistra a destra. Restituisce 1 se entrambi i suoi operandi risultano diversi da zero, altrimenti 0. A differenza di **&**, **&&** garantisce la valutazione da sinistra a destra: si valuta il primo operando, compresi tutti gli effetti collaterali; se è uguale a 0, il valore dell'espressione è 0. Se così non è, si valuta l'operando destro; se è uguale a 0, il valore dell'espressione è 0, altrimenti è 1.

Non è necessario che gli operandi abbiano lo stesso tipo, ma ognuno deve avere tipo aritmetico o essere un puntatore. Il risultato è **int**.

### A.7.15 Operatore OR logico

*espressione-OR-logico:*

espressione-AND-logico

espressione-OR-logico || espressione-AND-logico

L'operatore **||** si raggruppa da sinistra a destra. Se uno dei suoi operandi risulta diverso da zero, restituisce 1, diversamente 0. A differenza di **!**, **||** assicura la valutazione da sinistra a destra: si valuta il primo operando, inclusi tutti gli effetti collaterali; se è diverso da 0, il valore dell'espressione è 1. Altrimenti, si valuta l'operando di destra, e se è diverso da zero, il valore dell'espressione è 1, altrimenti è 0.

Non è necessario che gli operandi abbiano lo stesso tipo, ma ognuno deve avere tipo aritmetico o essere un puntatore. Il risultato è **int**.

### A.7.16 Operatore condizionale

*espressione-condizionale:*

espressione-OR-logico

espressione-OR-logico ? espressione : espressione-condizionale

La prima espressione è valutata, compresi gli effetti collaterali; se è diversa da 0, il risultato è il valore della seconda espressione, in caso contrario è quello della terza espressione. Si valuta solo il secondo o il terzo operando. Se il secondo e il terzo operando sono aritmetici, le conversioni aritmetiche abituali vengono effettuate per trasformarli in un tipo comune, e quello è il tipo del risultato. Qualora siano entrambi void, oppure strutture o unioni dello stesso tipo, o puntatori a oggetti dello stesso tipo, il risultato mantiene il tipo comune. Se uno è un puntatore e l'altro la costante 0, lo 0 viene convertito nel tipo del puntatore, e il risultato avrà quel tipo. Se uno è un puntatore a void mentre l'altro è un puntatore diverso, quest'ultimo è convertito in un puntatore a void, che sarà anche il tipo del risultato.

Nel confronto fra tipi per puntatori, i qualificatori di tipo (§A.8.2), quali che siano, sono irrilevanti nel tipo a cui punta il puntatore, ma il tipo del risultato eredita i qualificatori di entrambi i rami della proposizione condizionale.

### A.7.17 Espressioni di assegnamento

Esistono numerosi operatori di assegnamento, e tutti si raggruppano da destra a sinistra.

*espressione-assegnamento:*

espressione-condizionale

espressione-unaria operatore-assegnamento espressione-assegnamento

*operatore-assegnamento:* uno fra

= \*= /= %= += -= <<= >>= &= ^= |=

Tutti richiedono come operando di sinistra un lvalue, che deve essere modificabile: non deve essere un vettore, e non deve avere un tipo incompleto o essere una funzione. Inoltre, il suo tipo non deve essere qualificato da **const**; se è una struttura o unione, non deve avere

alcun membro o, ricorsivamente, sottomembro qualificato da `const`. Il tipo di un'espressione di assegnamento è quello del proprio operando sinistro, e il suo valore è il valore memorizzato nell'operando sinistro dopo l'esecuzione dell'assegnamento.

Nel semplice assegnamento con `=`, il valore dell'espressione sostituisce quello dell'oggetto a cui si riferisce lo `lvalue`. Una delle seguenti asserzioni deve essere vera: ambedue gli operandi hanno tipo aritmetico, nel qual caso l'operando destro è convertito dall'assegnamento nel tipo di sinistra; oppure entrambi gli operandi sono strutture o unioni dello stesso tipo; oppure un operando è un puntatore e l'altro è un puntatore a `void`; oppure l'operando sinistro è un puntatore mentre il destro è un'espressione costante di valore 0; o, infine, entrambi gli operandi sono puntatori a funzioni o a oggetti che abbiano lo stesso tipo a eccezione dell'eventuale mancanza di `const` o `volatile` nell'operando di destra.

Un'espressione della forma `E1 op= E2` è equivalente a `E1 = E1 op (E2)`, tranne per il fatto che `E1` viene calcolata una sola volta.

### A.7.18 Operatore virgola

*espressione:*

*espressione-assegnamento*

*espressione , espressione-assegnamento*

Una coppia di espressioni separate da una virgola è valutata da sinistra a destra, e il valore dell'espressione di sinistra è scartato. Il tipo e il valore del risultato sono quelli dell'operando destro. Tutti gli effetti collaterali che derivano dalla valutazione dell'operando sinistro sono portati a termine prima di iniziare la valutazione dell'operando destro. In contesti in cui alla virgola è attribuita una valenza particolare, per esempio nelle liste di argomenti di una funzione (§A.7.3.2) e nelle liste di inizializzatori (§A.8.7), l'unità sintattica necessaria è un'espressione di assegnamento, e quindi l'operatore virgola appare esclusivamente in un raggruppamento tra parentesi; per esempio,

```
f(a, (t=3, t+2), c)
```

ha tre argomenti, il secondo dei quali ha valore 5.

### A.7.19 Espressioni costanti

Sintatticamente, un'espressione costante è un'espressione limitata a un sottoinsieme di operatori:

*espressione-costante:*

*espressione-condizionale*

Espressioni la cui valutazione produca una costante sono richieste in numerosi contesti: dopo `case`, come limiti di vettori e lunghezze dei campi di bit, come valore di una costante di enumerazione, negli inizializzatori e in talune espressioni del preprocessore.

Le espressioni costanti non possono contenere assegnamenti, operatori di incremento o decremento, chiamate a funzioni e operatori virgola, eccetto in un operando di `sizeof`. Se

l'espressione costante deve essere integrale, i suoi operandi devono consistere di costanti intere, di enumerazione, carattere e con virgola mobile; i cast devono specificare un tipo integrale, e tutte le costanti con virgola mobile devono essere trasformate in interi tramite cast. Ciò esclude necessariamente i vettori, l'indirezione, il reperimento degli indirizzi e l'accesso ai membri delle strutture. (Tuttavia `sizeof` ammette qualsiasi operando.)

Godono di una maggiore elasticità le espressioni costanti degli inizializzatori; gli operandi possono essere qualsiasi tipo di costante, e l'operatore unario `&` può essere applicato a oggetti esterni o statici, così come a vettori esterni o statici indicati da un'espressione costante. L'operatore unario `&` è altresì implicitamente adoperato nei nomi di vettori privi di indici e nei nomi delle funzioni. La valutazione degli inizializzatori deve risultare in una costante oppure nell'indirizzo di un oggetto esterno o statico precedentemente dichiarato, più o meno una costante.

Minore è la tolleranza nei confronti delle espressioni costanti integrali dopo `#if`; le espressioni `sizeof`, le costanti di enumerazione e i cast non sono permessi. Si veda §A.12.5.

## A.8 Dichiarazioni

Le dichiarazioni precisano l'interpretazione data a ogni identificatore; non stanziano necessariamente memoria associata all'identificatore. Le dichiarazioni che allocano memoria sono chiamate *definizioni*. Le dichiarazioni hanno la forma:

*dichiarazione:*

*specificatori-dichiarazione lista-iniz-dichiaratore<sub>fac</sub> ;*

I dichiaratori in *lista-iniz-dichiaratore* contengono gli identificatori da dichiarare; gli specificatori-dichiarazione sono formati da una serie di specificatori relativi al tipo e alla classe di memorizzazione.

*specificatori-dichiarazione:*

*specificatore-classe-memoria specificatori-dichiarazione<sub>fac</sub>*

*specificatore-tipo specificatori-dichiarazione<sub>fac</sub>*

*qualificatore-tipo specificatori-dichiarazione<sub>fac</sub>*

*lista-iniz-dichiaratore:*

*iniz-dichiaratore*

*lista-iniz-dichiaratore , iniz-dichiaratore*

*iniz-dichiaratore:*

*dichiaratore*

*dichiaratore = inizializzatore*

I dichiaratori saranno approfonditi più avanti (§A.8.5); essi contengono i nomi da dichiarare. Una dichiarazione deve avere almeno un dichiaratore, oppure il suo specificatore di tipo deve dichiarare il contrassegno di una struttura, o di un'unione, o i membri di un'enumerazione; le dichiarazioni vuote non sono permesse.

### A.8.1 Specificatori della classe di memorizzazione

Gli specificatori della classe di memorizzazione sono:

*specificatore-classe-memoria:*

- auto
- register
- static
- extern
- typedef

I significati delle classi di memorizzazione sono stati trattati in §A.4.

Gli specificatori `auto` e `register` attribuiscono agli oggetti dichiarati una classe di memorizzazione automatica, e possono essere usati solo all'interno di funzioni. Tali dichiarazioni fungono anche da definizioni e determinano lo stanziamento di memoria. Una dichiarazione `register` è equivalente a una dichiarazione `auto`, ma suggerisce che si accederà spesso agli oggetti dichiarati. Sono pochi, in realtà, gli oggetti che vengono collocati nei registri, e solo alcuni tipi possiedono i requisiti necessari; le restrizioni dipendono dall'implementazione. Tuttavia, a un oggetto dichiarato `register` non si può applicare l'operatore unario `&`, né esplicitamente né implicitamente.

La regola per cui è illegale calcolare l'indirizzo di un oggetto dichiarato `register`, ma che sia in realtà `auto`, è nuova.

Lo specificatore `static` attribuisce agli oggetti dichiarati la classe di memorizzazione `static`, e può essere utilizzato sia all'interno che all'esterno delle funzioni. All'interno di una funzione, questo specificatore fa sì che la memoria sia allocata, e funge da definizione; rimandiamo a §A.11.2 per il suo effetto al di fuori di una funzione.

Una dichiarazione con `extern`, usata all'interno di una funzione, specifica che la memoria per gli oggetti dichiarati è definita altrove; quanto ai suoi effetti al di fuori di una funzione, si veda §A.11.2.

Lo specificatore `typedef` non stanzia memoria ed è classificato come specificatore di classe di memorizzazione solo per convenienza sintattica; è trattato in §A.8.9.

Non è ammesso più di uno specificatore di classe di memorizzazione per ogni dichiarazione. Nel caso non ve ne siano del tutto, sono usate queste regole: gli oggetti dichiarati all'interno di una funzione sono considerati `auto`; le funzioni dichiarate all'interno di una funzione sono considerati `extern`; gli oggetti e le funzioni dichiarate al di fuori di una funzione sono considerati `static`, con linkage esterno. Si veda §§A.10-A.11.

### A.8.2 Specificatori di tipo

Gli specificatori di tipo sono

*specificatore-tipo:*

- void
- char

short  
int  
long  
float  
double  
signed  
unsigned  
*specificatore-strutt-o-unione*  
*specificatore-enum*  
*nome-typedef*

Una sola parola tra `short` e `long` può essere specificata insieme a `int`; il significato rimane identico se `int` non è menzionato. La parola `long` può essere specificata insieme a `double`. Solo uno tra `signed` e `unsigned` può essere specificato con `int` o con una delle sue varianti `short` o `long`, o con `char`. Possono entrambi apparire da soli, e in questo caso si sottintende `int`. Lo specificatore `signed` è utile per imporre il segno agli oggetti `char`; è ammesso ma ridondante con altri tipi integrali.

Negli altri casi, non più di uno specificatore di tipo può essere fornito in una dichiarazione. Se lo specificatore di tipo manca in una dichiarazione, viene considerato `int`.

Anche i tipi possono essere qualificati da attributi, al fine di evidenziare proprietà rilevanti degli oggetti che si dichiarano.

*qualificatore-tipo:*

- const
- volatile

I qualificatori di tipo possono accompagnare qualunque specificatore di tipo. Un oggetto `const` può essere inizializzato, ma poi non più assegnato. Per gli oggetti `volatile` la semantica dipende totalmente dall'implementazione.

Le proprietà di `const` e `volatile` sono una novità dello standard ANSI. Il qualificatore `const` serve a dichiarare oggetti che possono trovare collocazione nella memoria a sola lettura, e forse ad aumentare le possibilità di ottimizzazione. La finalità di `volatile` consiste nell'imporre un'implementazione che elimini le ottimizzazioni altrimenti possibili. Per esempio, per una macchina con l'input/output mappato in memoria, un puntatore al registro di un dispositivo potrebbe essere dichiarato come puntatore a `volatile`, così da evitare che il compilatore elimini riferimenti apparentemente ridondanti tramite il puntatore. I compilatori possono ignorare questi qualificatori, ma devono segnalare esplicativi tentativi di modifica degli oggetti `const`.

### A.8.3 Dichiarazioni di struttura e di unione

Una struttura è un oggetto formato da una serie di vari tipi di membri provvisti di nome. Un'unione è un oggetto che contiene, a seconda dei momenti, uno fra diversi membri appartenenti a vari tipi. Gli specificatori di struttura e di unione hanno la stessa forma.

*specificatore-strutt-o-unione:*

```
strutt-o-unione identificatorefac { lista-dichiarazione-strutt }
    strutt-o-unione identificatore
```

*strutt-o-unione:*

```
struct
union
```

Una lista-dichiarazione-strutt è una sequenza di dichiarazioni per i membri della struttura o dell'unione:

*lista-dichiarazione-strutt:*

```
dichiarazione-strutt
lista-dichiarazione-strutt dichiarazione-strutt
```

*dichiarazione-strutt:*

```
lista-specificatore-qualificatore lista-dichiaratore-strutt ;
```

*lista-specificatore-qualificatore:*

```
specificatore-tipo lista-specificatore-qualificatorefac
    qualificatore-tipo lista-specificatore-qualificatorefac
```

*lista-dichiaratore-strutt:*

```
dichiaratore-strutt
lista-dichiaratore-strutt , dichiaratore-strutt
```

Di norma, un dichiaratore-strutt è un semplice dichiaratore per un membro di una struttura o unione. Il membro di una struttura può anche consistere di un determinato numero di bit. Un membro siffatto si chiama anche *campo di bit*, o in breve *campo*; la sua lunghezza è separata dal dichiaratore del nome del campo mediante i due punti.

*dichiaratore-strutt:*

```
dichiaratore
dichiaratorefac : espressione-costante
```

Uno specificatore di tipo della forma

```
strutt-o-unione identificatore { lista-dichiarazione-strutt }
```

dichiara che l'identificatore è il *contrassegno* (*tag*) della struttura o dell'unione specificata dalla lista. Una dichiarazione successiva che si trovi nel medesimo campo di visibilità, oppure in uno più interno, può riferirsi allo stesso tipo usando il contrassegno in uno specificatore senza la lista:

```
strutt-o-unione identificatore
```

Se uno specificatore con un contrassegno ma privo di lista appare quando il contrassegno non è dichiarato, è specificato un *tipo incompleto*. Gli oggetti con un tipo incompleto di struttura o di unione possono essere citati esclusivamente nei contesti in cui la loro dimensione non è necessaria, per esempio in dichiarazioni (non definizioni), per specificare

un puntatore, o per creare un *typedef*. Il tipo diventa completo quando interviene un altro specificatore con quel contrassegno, e contenente una lista di dichiarazioni. Persino negli specificatori con una lista, il tipo della struttura o dell'unione che si sta dichiarando è incompleto all'interno della lista, e diviene completo solo al raggiungimento della } di chiusura.

Una struttura non può contenere un membro di tipo incompleto. Conseguentemente è impossibile dichiarare una struttura o unione che contenga un'istanza di se stessa. Ciononostante, oltre ad attribuire un nome al tipo della struttura o dell'unione, i contrassegni consentono la definizione di strutture autoreferenziali; una struttura o un'unione può contenere un puntatore a una struttura del suo stesso tipo, poiché sono ammesse le dichiarazioni di puntatori a tipi incompleti.

Una regola eccezionale si applica alle dichiarazioni della forma

```
strutt-o-unione identificatore ;
```

che dichiarano una struttura o unione, nonostante siano prive sia della lista di dichiarazione che dei dichiaratori. Anche se l'identificatore è il contrassegno per una struttura o unione già dichiarato in un campo di visibilità più esterno (§A.11.1), questa dichiarazione trasforma l'identificatore nel contrassegno di una nuova struttura o unione che ha tipo incompleto e risiede nel campo di visibilità attuale.

Questa astrusa regola è una novità dello standard ANSI. È dedicata alle strutture mutualmente ricorsive dichiarate in un campo di visibilità più interno, ma i cui contrassegni possano essere già stati dichiarati in un campo di visibilità più esterno.

Uno specificatore di unione o di struttura dotato di lista ma non di contrassegno crea un tipo unico; a esso ci si può riferire direttamente solo nella dichiarazione di cui fa parte.

I nomi dei membri e dei contrassegni non entrano in conflitto tra loro o con le variabili ordinarie. Il nome di un membro non può comparire due volte nella stessa struttura o unione, ma lo stesso nome può essere usato in strutture o unioni differenti.

Nella prima edizione di questo libro, i nomi dei membri delle strutture e delle unioni non erano associati al loro genitore. Tuttavia, questa associazione divenne una prassi per i compilatori ben prima dello standard ANSI.

Un membro di una struttura o di un'unione che non sia un campo può avere qualsiasi tipo di oggetto. Un membro che sia un campo (che non ha bisogno di un dichiaratore e può dunque essere anonimo) ha per tipo *int*, *unsigned int* o *signed int*, ed è interpretato come oggetto di tipo integrale della lunghezza specificata in bit; dipende dall'implementazione se un campo *int* sia considerato dotato di segno. Campi adiacenti nelle strutture sono impaccati in unità di memoria dipendenti dall'implementazione in una direzione che dipende anch'essa dall'implementazione. Quando un campo ne segue un altro, e non è di grandezza idonea a occupare un'unità di memoria parzialmente piena, può essere scomposto in diverse unità, o l'unità in questione può essere riempita artificialmente. Un campo anonimo di ampiezza 0 forza tale operazione di riempimento, per cui il campo successivo sarà allineato all'unità di allocazione successiva.

Lo standard ANSI vincola i campi all'implementazione persino più strettamente di quanto facesse la prima edizione. È consigliabile interpretare le regole del linguaggio per la memorizzazione dei campi di bit come "dipendenti dall'implementazione" tout court. Le strutture con campi di bit possono essere utilizzate come tentativi portabili di ridurre lo spazio di memoria richiesto dalla struttura (con un probabile aumento dello spazio e del tempo richiesti dalle istruzioni per l'accesso ai campi), o come metodo non portabile per descrivere un'architettura di memoria conosciuta a livello di bit. Nell'ultima ipotesi, è necessario capire le regole dell'implementazione locale.

Gli indirizzi dei membri di una struttura sono crescenti nell'ordine di dichiarazione dei membri. Un membro che non sia un campo della struttura è allineato a un punto indirizzabile della memoria dipendente dal suo tipo; pertanto, possono esservi buchi anonimi in una struttura. Se un puntatore a una struttura è trasformato tramite cast nel tipo di un puntatore al primo membro della stessa, il risultato si riferisce al primo membro.

Un'unione può essere vista come una struttura in cui tutti i membri abbiano uno scostamento nullo e la cui dimensione sia sufficiente a contenere qualunque suo membro. Non più di un membro può essere memorizzato all'interno di un'unione in un dato istante. Se un puntatore a un'unione è trasformato tramite un cast nel tipo di un puntatore a un membro, il risultato si riferirà a tale membro.

Un esempio elementare di dichiarazione di una struttura è

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
};
```

che contiene un vettore di 20 caratteri, un intero e due puntatori a strutture simili. Una volta che questa dichiarazione sia stata fatta, la dichiarazione

```
struct tnode s, *sp;
```

dichiara che *s* è una struttura del genere specificato e che *sp* è un puntatore a una struttura del genere specificato. Con queste dichiarazioni, l'espressione

```
sp->count
```

è riferita al campo *count* della struttura a cui punta *sp*;

```
s.left
```

si riferisce al puntatore al sottoalbero sinistro della struttura *s*;

```
s.right->tword[0]
```

si riferisce al primo carattere del membro *tword* del sottoalbero destro di *s*.

Generalmente un membro di un'unione non può essere ispezionato a meno che il valore dell'unione sia stato assegnato usando quello stesso membro. Tuttavia, una garanzia speciale semplifica l'utilizzo delle unioni: se un'unione contiene molteplici strutture che condividono una sequenza iniziale comune, e posto che l'unione contenga una di queste

strutture, è permesso riferirsi alla parte iniziale che è in comune fra tutte le strutture. A seguire, un esempio di brano legale:

```
union {
    struct {
        int type;
    } n;
    struct {
        int type;
        int intnode;
    } ni;
    struct {
        int type;
        float floatnode;
    } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
    ... sin(u.nf.floatnode) ...
```

#### A.8.4 Enumerazioni

Le enumerazioni sono tipi unici che assumono valori in un insieme di costanti provviste di nome chiamate enumeratori. La forma di uno specificatore di enumerazione è plasmata su quella delle strutture e delle unioni.

```
specificatore-enum:
    enum identificatorefac { lista-enumeratore }
    enum identificatore

lista-enumeratore:
    enumeratore
    lista-enumeratore , enumeratore

enumeratore:
    identificatore
    identificatore = espressione-costante
```

Gli identificatori in una lista di enumeratori sono dichiarati come costanti di tipo *int*, e possono comparire dovunque le costanti siano necessarie. Se non si danno enumeratori con *=*, i valori delle costanti corrispondenti cominciano da 0 e aumentano di 1 mentre la dichiarazione è letta da sinistra a destra. Un enumeratore con *=* assegna all'identificatore associato il valore specificato; gli identificatori successivi continuano la progressione dal valore assegnato.

I nomi degli enumeratori nello stesso campo di visibilità devono essere tutti distinti fra di loro e dai nomi delle variabili ordinarie, mentre i loro valori non devono essere distinti.

Il ruolo dell'identificatore nello specificatore-enum è analogo al ruolo del contrassegno della struttura in uno specificatore-strutt; esso nomina una particolare enumerazione. Le regole di specificatore-enum, con o senza contrassegno e lista, corrispondono a quelle degli specificatori di struttura e di unione, eccetto che non esistono tipi enumerativi incompleti; il contrassegno di uno specificatore-enum privo di una lista di enumeratori deve riferirsi a uno specificatore con lista visibile.

Le enumerazioni sono una novità rispetto alla prima edizione di questo libro, anche se fanno parte del linguaggio da alcuni anni.

## A.8.5 Dichiariatori

I dichiariatori hanno la sintassi:

dichiariatore:

$puntatore_{fac} \text{ dichiaratore-diretto}$

dichiariatore-diretto:

identificatore

( dichiaratore )

dichiariatore-diretto [ espressione-costante<sub>fac</sub> ]

dichiariatore-diretto ( lista-tipo-parametro )

dichiariatore-diretto ( lista-identificatore<sub>fac</sub> )

puntatore:

\* lista-qualificatore-tipo<sub>fac</sub>

\* lista-qualificatore-tipo<sub>fac</sub> puntatore

lista-qualificatore-tipo:

qualificatore-tipo

lista-qualificatore-tipo qualificatore-tipo

La struttura dei dichiariatori assomiglia a quella delle espressioni relative all'indirezione, alle funzioni e ai vettori: le regole per il raggruppamento sono le stesse.

## A.8.6 Significato dei dichiariatori

Una lista di dichiariatori compare dopo una sequenza di specificatori di tipo e di classe di memorizzazione. Ogni dichiaratore dichiara un solo identificatore principale, quello che appare come prima alternativa della produzione per *dichiariatore-diretto*. Gli specificatori della classe di memorizzazione si applicano direttamente a questo identificatore, ma il suo tipo dipende dalla forma del relativo dichiaratore. Un dichiaratore è inteso come asserzione che, quando il suo identificatore appare in un'espressione della stessa forma del dichiaratore, produce un oggetto del tipo specificato.

Considerando solo le parti attinenti al tipo degli specificatori della dichiarazione (§A.8.2) e un dichiaratore particolare, una dichiarazione assume la forma “*T D*”, dove *T* è un tipo e *D* è un dichiaratore. Il tipo attribuito all'identificatore nelle varie forme del dichiaratore è descritto induttivamente usando questa notazione.

In una dichiarazione *T D*, dove *D* è un identificatore semplice, il tipo dell'identificatore è *T*.

In una dichiarazione *T D* dove *D* ha la forma

( *D1* )

il tipo dell'identificatore in *D1* è lo stesso di quello di *D*. Le parentesi non modificano il tipo, ma possono cambiare i diritti di precedenza nei dichiaratori complessi.

### A.8.6.1 Dichiariatori dei puntatori

In una dichiarazione *T D* dove *D* ha la forma

\* lista-qualificatore-tipo<sub>fac</sub> *D1*

e il tipo dell'identificatore nella dichiarazione *T D1* è “*modificatore-tipo T*”, il tipo dell'identificatore di *D* è “*modificatore-tipo lista-qualificatore-tipo puntatore a T*”. I qualificatori che seguono \* si applicano al puntatore stesso, anziché all'oggetto puntato.

Come esempio, si prenda in esame la dichiarazione

```
int *ap[];
```

*ap[ ]* gioca il ruolo di *D1*; la dichiarazione “*int ap[ ]*” (si veda di seguito) darebbe ad *ap* il tipo “vettore di *int*”, la lista dei qualificatori di tipo è vuota, e il modificatore di tipo è “vettore di”. Quindi la dichiarazione in questione dota *ap* del tipo “vettore di puntatori a *int*”.

Altri esempi sono le dichiarazioni

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

che dichiarano un intero *i* e un puntatore a un intero *pi*. Il valore del puntatore costante *cpi* non può essere modificato, e punterà sempre alla stessa locazione, sebbene il valore a cui si riferisce possa essere cambiato. L'intero *ci* è costante, e non può essere modificato (benché possa essere inizializzato, come in questo caso). Il tipo di *pci* è “puntatore a *const int*”, e *pci* medesimo può subire modifiche per puntare a un'altra posizione; ciò non vale per il valore a cui punta, il quale non può essere alterato da assegnamenti che coinvolgono *pci*.

### A.8.6.2 Dichiariatori di vettore

In una dichiarazione *T D* dove *D* assume la forma

*D1* [ espressione-costante<sub>fac</sub> ]

e il tipo dell'identificatore nella dichiarazione *T D1* è “*modificatore-tipo T*”, il tipo dell'identificatore di *D* è “*modificatore-tipo vettore di T*”. Se l'espressione-costante è presente, deve avere tipo integrale, e valore maggiore di 0. Se manca l'espressione costante che indica la dimensione, il vettore ha un tipo incompleto.

Un vettore può essere costruito a partire da un tipo aritmetico, da un puntatore, da una struttura o unione, oppure da un altro vettore (per generare un vettore multidimensionale). Il tipo da cui si ricava un vettore deve essere completo, e non può essere un vettore o una struttura di tipo incompleto. Ciò implica che in un vettore multidimensionale può mancare solo la prima dimensione. Il tipo di un oggetto di tipo vettore incompleto è completato con un'altra dichiarazione, completa, dell'oggetto (§A.10.2) o inizializzandolo (§A.8.7).

Per esempio,

```
float fa[17], *afp[17];
```

dichiara un vettore di numeri float e un vettore di puntatori a numeri float. Anche

```
static int x3d[3][5][7];
```

dichiara un vettore statico tridimensionale di interi, che ha 3x5x7 elementi. Entrando nei dettagli, x3d è un vettore di tre elementi; ogni elemento è un vettore di cinque vettori, ognuno dei quali è un vettore di sette interi. Ciascuna delle espressioni x3d, x3d[i], x3d[i][j], x3d[i][j][k] può ragionevolmente comparire in un'espressione. Le prime tre hanno per tipo "vettore", l'ultima ha per tipo int. Più precisamente, x3d[i][j] è un vettore di 7 interi, e x3d[i] è un vettore di 5 vettori di 7 interi.

L'accesso tramite indici a elementi di un vettore è definito in modo tale che E1[E2] sia un'espressione identica a \*(E1+E2). Di conseguenza, malgrado il suo aspetto asimmetrico, l'accesso tramite indici è un'operazione commutativa. A causa delle regole di conversione che valgono per + e per i vettori (§§A.6.6, A.7.1, A.7.7), se E1 è un vettore e E2 un intero, E1[E2] si riferisce all'E2-esimo membro di E1.

Nell'esempio, x3d[i][j][k] è equivalente a \*(x3d[i][j] + k). La prima sottoespressione x3d[i][j] è convertita, in forza di §A.7.1, nel tipo "puntatore a vettore di interi"; stante §A.7.7, l'addizione comporta la moltiplicazione per la grandezza di un intero. Per effetto delle regole, i vettori sono memorizzati per righe (l'ultimo indice varia più velocemente) e il primo indice nella dichiarazione aiuta a individuare la quantità di memoria consumata da un vettore, ma non ha altro ruolo nel calcolo degli indici.

#### A.8.6.3 Dichiaratori di funzione

In una dichiarazione di funzione T D secondo la nuova modalità, dove D ha la forma

```
D1(lista-tipo-parametro)
```

e il tipo dell'identificatore nella dichiarazione T D1 è "modificatore-tipo T", il tipo dell'identificatore di D è "modificatore-tipo funzione con argomenti lista-tipo-parametro che restituisce T".

La sintassi dei parametri è

*lista-tipo-parametro*:

```
lista-parametro  
lista-parametro , ...
```

*lista-parametro*:

```
dichiarazione-parametro  
lista-parametro , dichiarazione-parametro
```

*dichiarazione-parametro*:

```
specificatori-dichiarazione dichiaratore  
specificatori-dichiarazione dichiaratore-astrattofac
```

Nelle dichiarazioni secondo la nuova modalità, la lista dei parametri specifica i tipi dei parametri. Come caso particolare, il dichiaratore di una funzione che segue la nuova modalità

senza parametri ha una lista-tipo-parametro composta solo dalla parola chiave void. Se la lista-tipo-parametro termina con un'ellissi "...", la funzione può accettare più argomenti del numero di parametri esplicitamente descritto; si veda §A.7.3.2.

I tipi dei parametri che sono vettori o funzioni sono mutati in puntatori, secondo le regole per la conversione dei parametri; si veda §A.10.1. L'unico specificatore della classe di memorizzazione permesso come specificatore della dichiarazione di un parametro è register, e questo specificatore è ignorato a meno che il dichiaratore della funzione preceda la definizione di una funzione. In maniera analoga, se i dichiaratori nelle dichiarazioni dei parametri contengono identificatori, e il dichiaratore della funzione non precede la definizione di una funzione, gli identificatori escono immediatamente dal campo di visibilità. I dichiaratori astratti, che non citano gli identificatori, sono trattati in §A.8.8.

In una dichiarazione di funzione T D secondo la vecchia modalità, dove D ha la forma

```
D1(lista-identificatorefac)
```

e il tipo dell'identificatore nella dichiarazione T D1 è "modificatore-tipo T", il tipo dell'identificatore di D è "modificatore-tipo funzione con argomenti non specificati che restituisce T".

I parametri (se presenti) hanno la forma

*lista-identificatore*:

```
identificatore  
lista-identificatore , identificatore
```

Nel dichiaratore di vecchia concezione, la lista degli identificatori deve essere assente a meno che il dichiaratore sia usato nell'intestazione della definizione di una funzione (§A.10.1). Nessuna informazione relativa ai tipi dei parametri è fornita dalla dichiarazione.

Per esempio, la dichiarazione

```
int f(), *fpi(), (*pfi)();
```

definisce una funzione f che restituisce un intero, una funzione fpi che restituisce un puntatore a un intero, e un puntatore pfi a una funzione che restituisce un intero. In nessuna di queste dichiarazioni, scritte secondo la vecchia modalità, sono specificati i tipi dei parametri.

Nella dichiarazione secondo la nuova modalità

```
int strcpy(char *dest, const char *source), rand(void);
```

strcpy è una funzione che restituisce int, con due argomenti, il primo un puntatore a carattere e il secondo un puntatore a caratteri costanti. I nomi dei parametri sono di fatto commenti. La seconda funzione rand non accetta argomenti e restituisce int.

I dichiaratori di funzione con prototipi dei parametri costituiscono decisamente la più importante innovazione apportata finora al linguaggio dallo standard ANSI. Offrono un vantaggio rispetto ai dichiaratori "vecchio stile" della prima edizione: il rilevamento degli errori e la conversione degli argomenti nelle chiamate alle funzioni. Ma per tutto ciò si paga lo scotto della confusione che si è creata durante la loro introduzione, e della necessità di conciliare entrambe le forme. È stato necessario, per il bene della compatibilità, qualche obbrobrio sintattico, in particolare void, che segnala esplicitamente le funzioni di nuova modalità prive di parametri.

Anche la notazione ellittica “, ...” per funzioni con numero di argomenti variabile è nuova e, unitamente alle macro nell'intestazione standard `<stdarg.h>`, dà rigore formale a un meccanismo che era ufficialmente proibito ma ufficiosamente tollerato nella prima edizione.

Queste notazioni sono state adattate dal linguaggio C++.

### A.8.7 Inizializzazione

Quando viene dichiarato un oggetto, il suo *iniz-dichiaratore* può specificare un valore iniziale per l'identificatore che si dichiara. L'inizializzatore è preceduto da `=`, ed è un'espressione o una lista di inizializzatori annidati tra parentesi graffe. Una lista può terminare con una virgola, una finezza utile a formattare elegantemente.

*inizializzatore*:

```
espressione-assegnamento
{ lista-inizializzatore }
{ lista-inizializzatore , }
```

*lista-inizializzatore*:

```
inizializzatore
list-inizializzatore , inizializzatore
```

Tutte le espressioni nell'inizializzatore di un oggetto o di un vettore statico devono essere espressioni costanti come descritto in §A.7.19. Le espressioni nell'inizializzatore di un oggetto o di un vettore `auto` o `register` devono ugualmente essere costanti se l'inizializzatore è una lista racchiusa tra graffe. Malgrado ciò, se l'inizializzatore di un oggetto automatico è un'espressione singola, non deve essere un'espressione costante, ma è sufficiente che sia di tipo appropriato per l'assegnamento all'oggetto.

La prima edizione non approvava l'inizializzazione di vettori, strutture e unioni automatici. Lo standard ANSI la consente, ma solo per mezzo di costrutti costanti, a meno che l'inizializzatore possa essere realizzato come espressione semplice.

Un oggetto statico non esplicitamente inizializzato lo è come se a esso (o ai propri membri) fosse assegnata la costante 0. Il valore iniziale di un oggetto automatico non esplicitamente inizializzato è indefinito.

L'inizializzatore di un puntatore o di un oggetto di tipo aritmetico è un'espressione singola, che può stare tra parentesi. L'espressione viene assegnata all'oggetto.

L'inizializzatore di una struttura è un'espressione dello stesso tipo oppure una lista di inizializzatori in parentesi graffe corrispondenti ai membri. I campi di bit non muniti di nome sono ignorati, e non vengono inizializzati. Se la lista contiene più membri che inizializzatori, i membri in eccesso sono inizializzati a zero. Non possono esservi più inizializzatori che membri.

L'inizializzatore di un vettore è una lista racchiusa tra parentesi graffe di inizializzatori corrispondenti ai suoi membri. Se la dimensione del vettore è sconosciuta, sarà il numero di inizializzatori a determinarla, e il relativo tipo diventa completo. Qualora il vettore abbia dimensione fissa, il numero di inizializzatori non può superare quello dei membri del vettore; gli eventuali membri in sovrannumero sono inizializzati a zero.

Come caso particolare, un vettore di caratteri può essere inizializzato da un letterale stringa; caratteri successivi della stringa inizializzano membri successivi del vettore. Analogamente, una stringa di caratteri estesi (§A.2.6) può inizializzare un vettore di tipo `wchar_t`. Il numero di caratteri della stringa, incluso il carattere nullo di chiusura, ne determina la grandezza, se questa è sconosciuta; se è già determinata, il numero di caratteri della stringa, all'infuori dell'ultimo (null), non deve eccedere la dimensione del vettore.

L'inizializzatore di un'unione è un'espressione singola dello stesso tipo o un'inizializzatore tra parentesi graffe del primo membro dell'unione.

La prima edizione non consentiva l'inizializzazione delle unioni. La regola del “primo membro” è farraginosa, ma d'altra parte potrebbe difficilmente essere generalizzata senza definire della nuova sintassi. Oltre a rendere possibile l'inizializzazione esplicita delle unioni, anche se in modo rudimentale, questa regola dell'ANSI definisce comunque la semantica delle unioni statiche non esplicitamente inizializzate.

Un *aggregato* è una struttura o un vettore. Se un aggregato contiene membri di tipo aggregato, le regole dell'inizializzazione sono applicate ricorsivamente. È possibile elidere le parentesi graffe nell'inizializzazione come segue: se l'inizializzatore di un membro dell'aggregato, che a sua volta è un aggregato, comincia con una parentesi graffa aperta, la seguente lista di inizializzatori separati da virgole inizializza i membri del sotto-aggregato; la presenza di un numero maggiore di inizializzatori rispetto ai membri costituisce un errore. Se, peraltro, l'inizializzatore di un sotto-aggregato non comincia con una parentesi graffa aperta, vengono prelevati dalla lista solo gli elementi sufficienti per inizializzare i membri del sotto-aggregato; gli eventuali membri residui sono destinati a inizializzare il successivo membro dell'aggregato in cui si trova il sotto-aggregato.

Per esempio,

```
int x[] = { 1, 3, 5 };
```

dichiara e inizializza `x` come vettore monodimensionale di tre membri, dato che la dimensione non è stata precisata e sono presenti tre inizializzatori.

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

è un'inizializzazione che contempla esplicitamente tutte le parentesi: 1, 3 e 5 inizializzano la prima riga del vettore `y[0]`, vale a dire `y[0][0], y[0][1], y[0][2]`. In modo analogo le due righe successive inizializzano `y[1]` e `y[2]`. L'inizializzatore termina in anticipo, e perciò gli elementi di `y[3]` sono inizializzati a 0. Esattamente lo stesso risultato si sarebbe potuto ottenere con

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

L'inizializzatore di *y* comincia con una parentesi graffa aperta, ma non così quello di *y[0]*; ecco perché vengono usati tre elementi della lista. I tre elementi seguenti sono estratti, per lo stesso motivo, prima per *y[1]*, quindi per *y[2]*. Inoltre,

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

inizializza la prima colonna di *y* (considerato come un vettore a due dimensioni) mentre lascia a 0 il resto.

Infine,

```
char msg[] = "Errore sintattico alla riga %s\n";
mostra un vettore di caratteri i cui membri sono inizializzati con una stringa; la sua dimensione comprende il carattere nullo conclusivo.
```

### A.8.8 Nomi di tipo

In diversi contesti (per specificare esplicitamente le conversioni di tipo con un cast, per dichiarare i tipi dei parametri nei dichiaratori delle funzioni, e come argomento di `sizeof`) è necessario fornire il nome di un tipo di dati. A ciò si perviene usando un *nome di tipo*, che è, sotto il profilo sintattico, una dichiarazione di un oggetto di quel tipo, priva del nome dell'oggetto.

*nome-tipo*:

*lista-specificatore-qualificatore dichiaratore-astratto<sub>fac</sub>*

*dichiaratore-astratto*:

*puntatore*

*puntatore<sub>fac</sub> dichiaratore-diretto-astratto*

*dichiaratore-diretto-astratto*:

( *dichiaratore-astratto* )

*dichiaratore-diretto-astratto<sub>fac</sub> [ espressione-costante<sub>fac</sub> ]*

*dichiaratore-diretto-astratto<sub>fac</sub> ( lista-tipo-parametro<sub>fac</sub> )*

È possibile identificare univocamente la posizione del dichiaratore-astratto in cui comparirebbe l'identificatore se l'espressione fosse un dichiaratore interno a una dichiarazione. Il tipo nominato è dunque lo stesso del tipo dell'identificatore ipotetico. Per esempio,

```
int
int *
int *[3]
int (*)[]
int }()
int (*[])(void)
```

nominano rispettivamente i tipi “intero”, “puntatore a intero”, “vettore di 3 puntatori a interi”, “puntatore a un vettore di un numero indeterminato di interi”, “funzione di parame-

tri imprecisi che restituiscono un puntatore a un intero” e “vettore, di grandezza indeterminata, di puntatori a funzioni senza parametri ognuna delle quali restituisce un intero”.

### A.8.9 Typedef

Le dichiarazioni il cui specificatore della classe di memorizzazione è `typedef` non dichiarano oggetti, ma definiscono identificatori che attribuiscono un nome ai tipi. Questi identificatori sono chiamati nomi `typedef`.

*nome-typedef*:  
    *identificatore*

Una dichiarazione `typedef` attribuisce un tipo a ciascun nome fra i propri dichiaratori, come di consueto (si veda il Paragrafo 8.6 del Capitolo 8). Da lì in avanti, ogni nome `typedef` di questo genere equivale sintatticamente a una parola chiave che specifichi il tipo relativo.

Per esempio, dopo

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

il codice

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

contiene dichiarazioni legali. Il tipo di *b* è `long`, quello di *bp* è “puntatore a `long`”, mentre quello di *z* è la struttura specificata; *zp* è un puntatore a una tale struttura.

Con `typedef` non si introducono tipi nuovi, bensì sinonimi per i tipi che possono già essere specificati in un altro modo. Nell'esempio, *b* ha lo stesso tipo di qualunque altro oggetto `long`.

I nomi `typedef` possono essere nuovamente dichiarati in un campo di visibilità più interno, ma necessitano in questo caso di un insieme di specificatori di tipo non vuoto. Per esempio,

```
extern Blockno;
```

non dichiara di nuovo `Blockno`, contrariamente a

```
extern int Blockno;
```

che lo fa.

### A.8.10 Equivalenza di tipo

Due liste di specificatori di tipo sono equivalenti se contengono lo stesso insieme di specificatori di tipo, tenendo presente che alcuni specificatori possono essere sottointesi da altri (per esempio, `long` sottintende `long int`). Le strutture, le unioni e le enumerazioni con diversi contrassegni sono distinte, e un'unione, una struttura o un'enumerazione priva di contrassegno denota un tipo unico.

Due tipi sono identici se i rispettivi dichiaratori astratti (§A.8.8), dopo aver sostituito eventuali tipi `typedef`, e cancellato gli identificatori dei parametri della funzione, sono uguali a meno di equivalenza delle liste degli specificatori di tipo. La dimensione dei vettori e i tipi dei parametri della funzione sono rilevanti.

## A.9 Istruzioni

Eccetto quando diversamente specificato, le istruzioni sono eseguite in sequenza. Le istruzioni sono eseguite in ragione del loro effetto, e non hanno valori. Esse ricadono in diverse categorie.

*istruzione:*

- istruzione-etichettata*
- istruzione-espressione*
- istruzione-composta*
- istruzione-scelta*
- istruzione-iterazione*
- istruzione-salto*

### A.9.1 Istruzioni etichettate

Le istruzioni possono recare etichette come prefissi.

*istruzione-etichettata:*

- identificatore : istruzione*
- case espressione-costante : istruzione*
- default : istruzione*

Un'etichetta che consista di un identificatore dichiara l'identificatore. L'unica applicazione di un'etichetta di identificatore è come destinazione di `goto`. Il campo di visibilità dell'identificatore è la funzione corrente. Poiché le etichette godono di uno spazio dei nomi a loro riservato, non interferiscono con altri identificatori e non possono essere dichiarate una seconda volta. Si consulta §A.11.1.

Le etichette dei casi, incluse quelle di `default`, sono usate con l'istruzione `switch` (§A.9.4).

L'espressione costante di `case` deve avere tipo integrale.

Le etichette, in sé e per sé, non alterano il flusso di controllo.

### A.9.2 Istruzioni espressione

Le istruzioni, in molti casi, sono istruzioni espressione, che assumono la forma

*istruzione-espressione:*

*espressionefac* ;

Le istruzioni espressione sono per lo più assegnamenti o chiamate alle funzioni. Tutti gli effetti collaterali dell'espressione sono completati prima che l'istruzione successiva sia eseguita.

ta. Se l'espressione è mancante, il costrutto è chiamato istruzione nulla, che è spesso usata per assegnare un corpo vuoto a un'iterazione o per collocare un'etichetta.

### A.9.3 Istruzioni composte

L'istruzione composta (detta anche "blocco") consente di usare numerose istruzioni laddove ne sarebbe richiesta una sola. Il corpo della definizione di una funzione è un'istruzione composta.

*istruzione-composta:*

{ *lista-dichiarazione<sub>fac</sub>* *lista-istruzione<sub>fac</sub>* }

*lista-dichiarazione:*

- dichiarazione*
- lista-dichiarazione dichiarazione*

*lista-istruzione:*

- istruzione*
- lista-istruzione istruzione*

Se nella lista-dichiarazione vi è un identificatore già visibile all'esterno del blocco, la dichiarazione esterna è sospesa all'interno del blocco (si veda §A.11.1), ma al di fuori di esso ritorna in vigore. Un identificatore può essere dichiarato solo una volta nello stesso blocco. Queste regole valgono per gli identificatori nello stesso spazio dei nomi (§A.11); gli identificatori che risiedono in differenti spazi dei nomi sono considerati distinti.

L'inizializzazione di oggetti automatici avviene ogni volta che si entra nel blocco dall'inizio, e procede nell'ordine dei dichiaratori. Se si accede al blocco da un punto intermedio, tramite un salto, tali processi di inizializzazione non saranno effettuati. L'inizializzazione di oggetti static avviene una sola volta, prima che il programma avvii l'esecuzione.

### A.9.4 Istruzioni di scelta

Le istruzioni di scelta selezionano uno fra diversi possibili flussi di controllo.

*istruzione-scelta:*

- if ( espressione ) istruzione*
- if ( espressione ) istruzione else istruzione*
- switch ( espressione ) istruzione*

In entrambe le forme dell'istruzione `if`, l'espressione, che deve avere tipo aritmetico o essere un puntatore, è valutata, effetti collaterali compresi; se risulta diversa a 0, si esegue la prima sotto-istruzione. Nella seconda forma, la seconda sotto-istruzione viene eseguita se l'espressione è 0. L'ambiguità derivante da `else` è risolta collegando la clausola `else` all'ultimo `if` privo di `else` incontrato allo stesso livello di annidamento.

L'istruzione `switch` determina il trasferimento del controllo a una fra varie istruzioni che dipendono dal valore di un'espressione, che deve avere tipo integrale. La sotto-istruzione controllata da uno `switch` è tipicamente composta. Qualunque istruzione nell'ambito della

sotto-istruzione può essere dotata di una o più etichette case (§A.9.1). L'espressione di controllo subisce la promozione integrale (§A.6.1), mentre le costanti relative alle clausole case sono convertite nel tipo promosso. Due costanti case dipendenti dallo stesso switch non possono avere lo stesso valore dopo la conversione. Inoltre può esservi al massimo un'etichetta default associata a uno switch. Le istruzioni switch possono essere annidate; un'etichetta case o default è associata al più piccolo switch che la contiene.

Quando l'istruzione switch è eseguita, la sua espressione viene valutata, effetti collaterali compresi, e confrontata con ogni costante delle clausole case. Se una di esse è uguale al valore dell'espressione, il controllo passa all'istruzione dell'etichetta case trovata; altrimenti, se è disponibile un'etichetta default, il controllo passa all'istruzione etichettata. Se nessuna opzione case è idonea, in mancanza di default non viene eseguita alcuna sotto-istruzione dipendente dall'istruzione switch.

Nella prima edizione dell'opera l'espressione di controllo di switch e le costanti relative alle clausole case dovevano avere tipo int.

### A.9.5 Istruzioni di iterazione

Le istruzioni di iterazione presiedono allo svolgimento dei cicli.

*istruzione-iterazione:*

```
while ( espressione ) istruzione
do istruzione while ( espressione ) ;
for ( espressionefac ; espressionefac ; espressionefac ) istruzione
```

Nelle istruzioni while e do, la sotto-istruzione è eseguita ripetutamente fintanto che il valore dell'espressione rimanga diverso da 0; l'espressione deve avere tipo aritmetico o essere un puntatore. Per while, la valutazione della condizione, compresi tutti gli effetti collaterali dell'espressione, ha luogo prima di ogni esecuzione dell'istruzione; per do, segue ciascuna ripetizione.

Nell'istruzione for, la prima espressione è valutata una volta, e di conseguenza determina l'inizializzazione del ciclo. Non vi è alcuna restrizione al suo tipo. La seconda espressione deve avere tipo aritmetico o essere un puntatore; viene valutata prima di ogni iterazione, e se diviene uguale a 0, il for si conclude. La terza espressione è valutata dopo ogni iterazione, e quindi specifica una nuova inizializzazione del ciclo. Al suo tipo non si applicano restrizioni. Gli effetti collaterali di ciascuna espressione sono portati a termine subito dopo la sua valutazione. Se la sotto-istruzione non ha al suo interno continue, un'istruzione

```
for ( espressione1 ; espressione2 ; espressione3 ) istruzione
```

è equivalente a

```
espressione1 ;
while ( espressione2 ) {
    istruzione
    espressione3 ;
}
```

Ciascuna delle tre espressioni può essere eliminata. Una seconda espressione mancante fa sì che la condizione sottintesa sia equivalente a testare una costante diversa da zero.

### A.9.6 Istruzioni di salto

Le istruzioni di salto trasferiscono il controllo senza condizioni.

*istruzione-salto:*

```
goto identificatore ;
continue ;
break ;
return espressionefac ;
```

Nell'istruzione goto, l'identificatore deve essere un'etichetta (§A.9.1) situata nella funzione corrente. Il controllo si sposta all'istruzione etichettata.

Un'istruzione continue può comparire solo all'interno di un'istruzione di iterazione, e determina il passaggio del controllo alla porzione di codice che prosegua il ciclo nella più piccola istruzione di iterazione che racchiuda il ciclo.

Più precisamente, nell'ambito di ciascuna delle istruzioni

while ( ... ) {	do {	for ( ... ) {
... contin: ;	... contin: ;	... contin: ;
}	} while ( ... );	}

un continue che non sia contenuto in una più piccola istruzione di iterazione è identico a goto contin.

Un'istruzione break può comparire solo all'interno di un'istruzione di iterazione o un'istruzione switch, e pone fine all'esecuzione della più piccola istruzione di questo tipo che la racchiuda; il controllo passa all'istruzione successiva a quella terminata.

Una funzione restituisce il controllo al suo chiamante con l'istruzione return. Quando return è seguito da un'espressione, il relativo valore è restituito al chiamante della funzione. L'espressione è convertita, come per assegnamento, al tipo restituito dalla funzione in cui appare.

Raggiungere la parentesi graffa di chiusura del corpo di una funzione è equivalente a un'istruzione return senza espressione. In ambedue le ipotesi, il valore restituito è indefinito.

## A.10 Dichiarazioni esterne

L'unità di input fornita al compilatore C è chiamata unità di traduzione; essa è formata da una serie di dichiarazioni esterne, che possono essere dichiarazioni oppure definizioni di funzione.

*unità-traduzione:*

```
dichiarazione-esterna
unità-traduzione dichiarazione-esterna
```

*dichiarazione-esterna:*  
*definizione-funzione*  
*dichiarazione*

Il campo di visibilità delle dichiarazioni esterne si estende fino al termine dell'unità di traduzione in cui sono dichiarate, così come le dichiarazioni interne ai blocchi hanno effetto fino al termine di essi. La sintassi delle dichiarazioni esterne non si discosta da quella delle altre dichiarazioni, ma il codice per le funzioni deve necessariamente risiedere a questo livello.

### A.10.1 Definizioni di funzione

Le definizioni di funzione hanno la forma

*definizione-funzione:*  
*specificatori-dichiarazione<sub>fac</sub> dichiaratore lista-dichiarazione<sub>fac</sub> istruzione-composta*

Gli unici specificatori della classe di memorizzazione consentiti tra gli specificatori della dichiarazione sono *extern* o *static*; si veda §A.11.2 per la relativa distinzione.

Una funzione può restituire un tipo aritmetico, una struttura, un'unione, un puntatore, o *void*, ma non una funzione o un vettore. Il dichiaratore in una dichiarazione della funzione deve affermare esplicitamente che il tipo dell'identificatore dichiarato è una funzione; ovvero, deve contenere una delle forme (si veda §A.8.6.3)

*dichiaratore-diretto ( lista-tipo-parametro )*  
*dichiaratore-diretto ( lista-identificatore<sub>fac</sub> )*

in cui il dichiaratore-diretto è un identificatore o un identificatore tra parentesi. In particolare, il dichiaratore non può generare il tipo di una funzione per mezzo di *typedef*.

Nella prima forma, la definizione è una funzione secondo la nuova modalità i cui parametri, insieme ai tipi, sono dichiarati nella relativa lista-tipo-parametro; la lista-dichiarazione che fa seguito al dichiaratore della funzione deve essere assente. A meno che la lista-tipo-parametro consista unicamente di *void*, indicando che la funzione non accetta alcun parametro, ciascun dichiaratore nella lista-tipo-parametro deve contenere un identificatore. Se la lista-tipo-parametro termina con “, ...” la funzione può essere chiamata con un numero di argomenti superiore a quello dei parametri; il meccanismo implementato dalla macro *va\_arg*, che si trova nell'intestazione standard *<stdarg.h>* ed è descritto nell'Appendice B, deve essere usato per riferirsi agli argomenti in eccedenza. Le funzioni con argomenti in numero variabile devono avere almeno un parametro munito di nome.

Nella seconda forma, la definizione segue la vecchia modalità: la lista di identificatori cita i parametri, mentre la lista di dichiarazione attribuisce loro i tipi. Se non vi è alcuna dichiarazione relativa a un dato parametro, si presuppone che il suo tipo sia *int*. La lista di dichiarazione deve dichiarare solo i parametri elencati nella lista, l'inizializzazione non è permessa, e il solo possibile specificatore della classe di memorizzazione è *register*.

In entrambi i modelli della definizione di una funzione, i parametri vanno dichiarati appena dopo l'inizio dell'istruzione composta che costituisce il corpo della funzione, motivo per cui i medesimi identificatori non devono essere dichiarati nuovamente a quel livello

(anche se possono essere ridichiarati nei blocchi interni, come tutti gli altri identificatori). Se si dichiara che un parametro ha tipo “vettore di *tipo*”, la dichiarazione è modificata in “puntatore a *tipo*”; analogamente, se si dichiara che un parametro ha tipo “funzione che restituisce *tipo*”, la dichiarazione è modificata in “puntatore a una funzione che restituisce *tipo*”. Nel corso di una chiamata a una funzione, gli argomenti sono convertiti come necessario e assegnati ai parametri; si consulti §A.7.3.2.

Le definizioni di funzione secondo la nuova modalità sono una novità per lo standard ANSI. Vi è anche una piccola differenza nei particolari di promozione; la prima edizione stabiliva che le dichiarazioni dei parametri *float* fossero modificate in *double*. La differenza diventa percepibile quando un puntatore a un parametro è generato all'interno di una funzione.

Un esempio compiuto di una definizione di funzione secondo la nuova modalità è

```
int max(int a, int b, int c)
{
    int m;

    m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

In questo caso *int* è lo specificatore della dichiarazione; *max(int a, int b, int c)* è il dichiaratore della funzione, e *{ ... }* è il blocco che fornisce il codice della funzione. La definizione corrispondente secondo la vecchia modalità sarebbe

```
int max(a, b, c)
int a, b, c;
{
    /* ... */
}
```

in cui *int max(a, b, c)* è adesso il dichiaratore, e *int a, b, c;* è la lista di dichiarazione per i parametri.

### A.10.2 Dichiarazioni esterne

Le dichiarazioni esterne specificano le caratteristiche di oggetti, funzioni e altri identificatori. Il termine “esterne” si riferisce alla loro ubicazione al di fuori delle funzioni, e non ha relazione diretta con la parola chiave *extern*; la classe di memorizzazione per un oggetto dichiarato esternamente può essere lasciata vuota, o può essere specificata come *extern* o *static*.

Svariate dichiarazioni esterne per lo stesso identificatore possono coesistere nella stessa unità di traduzione se concordano in relazione al tipo e al linkage, e se c'è al massimo una definizione per l'identificatore.

Si reputa che due dichiarazioni per un oggetto o una funzione concordino in quanto al tipo in base alle regole esposte in §A.8.10. Va aggiunto che se le dichiarazioni divergono perché il tipo di una è il tipo incompleto di una struttura, di un'unione o di un'enumera-

zione (§A.8.3), mentre quello dell'altra è il tipo corrispondente completo con lo stesso contrassegno, si conviene che i tipi concordino. E ancora, se uno è il tipo di un vettore incompleto (§A.8.6.2) e l'altro è il tipo di un vettore completo ma per tutto il resto sono identici, si conviene che essi concordino. Infine, se un tipo specifica una funzione secondo la vecchia modalità, e l'altro una funzione altrimenti identica, ma secondo la nuova modalità, con le dichiarazioni dei parametri, i tipi sono considerati identici.

Se la prima dichiarazione esterna di una funzione o di un oggetto include lo specificatore `static`, l'identificatore ha *linkage interno*; in caso contrario ha *linkage esterno*. Il linkage è trattato in §A.11.2.

Una dichiarazione esterna per un oggetto è una definizione se possiede un inizializzatore. Una dichiarazione esterna per un oggetto priva di inizializzatore, e di specificatore `extern`, è una *definizione provvisoria*. Se la definizione di un oggetto compare in un'unità di traduzione, eventuali definizioni provvisorie sono trattate come dichiarazioni ridondanti. Se l'unità di traduzione non contiene alcuna definizione dell'oggetto, tutte le sue definizioni provvisorie sono accorpate in una definizione singola con inizializzatore 0.

Ciascun oggetto deve avere una e una sola definizione. Per oggetti con linkage interno, questa regola si applica individualmente a ogni unità di traduzione, dato che tali oggetti sono unici nell'unità di traduzione. Per gli oggetti con linkage esterno, la regola si applica all'intero programma.

Sebbene la regola della definizione unica avesse una formulazione leggermente diversa nella prima edizione di questo libro, a tutti gli effetti è identica a quella appena esposta. Alcune implementazioni ne attenuano la forza generalizzando il concetto di definizione provvisoria. Nella formulazione alternativa, spesso riscontrata sui sistemi UNIX e riconosciuta come estensione comune dallo Standard, tutte le definizioni provvisorie di un oggetto con linkage esterno sono considerate simultaneamente in tutte le unità di traduzione del programma, anziché separatamente in ogni unità di traduzione. Qualora compaia una definizione in qualche punto del programma, le definizioni provvisorie diventeranno semplici dichiarazioni; se così non fosse, tutte le definizioni provvisorie verrebbero ricondotte a una sola definizione con inizializzatore 0.

## A.11 Campo di visibilità e linkage

Un programma non deve essere compilato tutto in una volta: il codice sorgente può essere conservato in diversi file contenenti le unità di traduzione, e le funzioni precompilate possono essere caricate dalle librerie. La comunicazione tra le funzioni di un programma può avvenire tramite chiamate che mediante elaborazione di dati esterni.

Vi sono, pertanto, due generi di campi di visibilità da esaminare: primo, il *campo di visibilità lessicale* di un identificatore, cioè la porzione di programma nei cui limiti sono recepite le caratteristiche dell'identificatore; secondo, la visibilità degli oggetti e delle funzioni con linkage esterno, che determina le connessioni tra identificatori in unità di traduzione compilate separatamente.

### A.11.1 Campo di visibilità lessicale

Gli identificatori ricadono in vari spazi dei nomi che non interferiscono reciprocamente; lo stesso identificatore può servire a diverse applicazioni, persino nel medesimo campo di visibilità, posto che tali utilizzi si svolgano in diversi spazi dei nomi. Le relative categorie sono: oggetti, funzioni, nomi `typedef` e costanti `enum`; etichette; contrassegni di strutture, unioni, ed enumerazioni; infine, i singoli membri di ciascuna struttura o unione.

Per molti versi tali regole si differenziano da quelle descritte nella prima edizione di questo manuale. Le etichette non avevano in precedenza un proprio spazio dei nomi; i contrassegni di strutture e unioni disponevano di spazi dei nomi distinti, e alcune implementazioni attribuivano uno spazio separato anche ai contrassegni di enumerazione; raccogliere nello stesso spazio contrassegni di natura diversa è una restrizione nuova. L'innovazione più importante rispetto alla prima edizione è che ogni struttura o unione crea uno spazio dei nomi separato per i suoi membri, cosicché lo stesso nome può apparire in numerose strutture diverse. Questa regola ha rappresentato una prassi comune per molti anni.

Il campo di visibilità lessicale di un identificatore di oggetto o di funzione in una dichiarazione esterna comincia dal termine del suo dichiaratore per estendersi all'intera unità di traduzione in cui compare. Il campo di visibilità di un parametro della definizione di una funzione comincia dall'inizio del blocco che definisce la funzione, e permane lungo tutta la funzione; il campo di visibilità di un parametro nella dichiarazione di una funzione termina alla fine del dichiaratore. Il campo di visibilità di un identificatore dichiarato in cima a un blocco inizia dal termine del suo dichiaratore, e permane fino alla fine del blocco. Un'etichetta è visibile nell'intera funzione in cui è contenuta. Il campo di visibilità del contrassegno di una struttura, unione o enumerazione, o di una costante enumerativa, comincia dalla sua occorrenza in uno specificatore di tipo, e permane per tutta l'unità di traduzione (per dichiarazioni al livello esterno) o fino alla fine del blocco (se le dichiarazioni sono all'interno di una funzione).

Se un identificatore è dichiarato esplicitamente in cima a un blocco, anche nel caso che questo costituisca una funzione, ogni dichiarazione dell'identificatore al di fuori del blocco perde la sua validità fino alla fine del blocco.

### A.11.2 Linkage

Nell'ambito dell'unità di traduzione, tutte le dichiarazioni di uno stesso identificatore di oggetto o funzione con linkage interno si riferiscono alla stessa cosa, e l'oggetto o funzione compete unicamente a quella unità di traduzione. Tutte le dichiarazioni dello stesso identificatore di oggetto o funzione con linkage esterno fanno riferimento alla stessa cosa, e l'oggetto o funzione è condiviso dall'intero programma.

Come visto in §A.10.2, la prima dichiarazione esterna di un identificatore gli attribuisce linkage interno se si usa lo specificatore `static`; diversamente il linkage è esterno. Se la dichiarazione per un identificatore all'interno di un blocco non comprende lo specificatore `extern`, l'identificatore non ha linkage e compete unicamente alla funzione. Qualora invece

essa contempla `extern`, e sia attiva una dichiarazione esterna dell'identificatore nel campo di visibilità che include il blocco, l'identificatore avrà lo stesso linkage della dichiarazione esterna, e farà riferimento allo stesso oggetto o funzione; ma in assenza di una dichiarazione esterna visibile, il suo linkage è esterno.

## A.12 Il preprocessore

Le operazioni di sostituzione delle macro, di compilazione condizionale e di inclusione dei file, sono affidate a un preprocessore. Le righe che hanno inizio con `#`, eventualmente preceduto da spazio bianco, interagiscono con il preprocessore. La sintassi di queste righe è indipendente dal resto del linguaggio; esse possono comparire dovunque e hanno effetto (non vincolato al campo di visibilità) per tutta l'unità di traduzione. I ritorni a capo sono significativi; ciascuna riga è analizzata individualmente (ma si veda §A.12.2 su come unire le righe). Per il preprocessore, un token è qualunque token del linguaggio, o una sequenza di caratteri che dia un nome di file, come nella direttiva `#include` (§A.12.4); inoltre, ogni carattere che non sia definito altrimenti è considerato un token. Tuttavia, l'effetto degli spazi, con esclusione degli spazi singoli e della tabulazione orizzontale, è indefinito nelle righe del preprocessore.

Le operazioni del preprocessore si svolgono in varie fasi che da un punto di vista logico sono successive, ma che possono essere accorpate da una specifica implementazione.

1. In primo luogo, le sequenze triplici descritte in §A.12.1 sono sostituite dai loro equivalenti. Se il sistema operativo dovesse richiederlo, tra le righe del sorgente sono inseriti i caratteri newline.
2. Ogni occorrenza di una barra inversa `\` seguita da un carattere newline è cancellata, facendo così congiungere le righe (§A.12.2).
3. Il programma è diviso in token separati da caratteri di spaziatura; i commenti sono sostituiti da uno spazio singolo. In seguito si applicano le direttive del preprocessore, e si espandono le macro (§§A.12.3-A.12.10).
4. Le sequenze di controllo nelle costanti carattere e stringa (§§A.2.5.2,A.2.6) sono sostituite dai loro equivalenti; quindi le costanti stringa adiacenti sono concatenate.
5. Il risultato è tradotto e poi integrato con altri programmi e librerie raccogliendo i programmi e i dati necessari, e collegando i riferimenti alle funzioni e agli oggetti esterni alle loro definizioni.

### A.12.1 Sequenze triple

Il set di caratteri del codice sorgente C è contenuto nella tabella ASCII a sette bit, ma è un sovrainsieme dellInvariant Code Set 646-1983. Perché i programmi possano essere rappresentati nel set ridotto, tutte le occorrenze delle seguenti sequenze triple sono sostituite dai caratteri singoli corrispondenti. Questa sostituzione avviene prima di ogni altra operazione.

<code>??=</code>	<code>#</code>	<code>??(</code>	<code>[</code>	<code>??&lt;</code>	<code>{</code>
<code>??\</code>		<code>??)</code>	<code>]</code>	<code>??&gt;</code>	<code>}</code>
<code>??^</code>	<code>~</code>	<code>??!</code>	<code> </code>	<code>??-</code>	<code>-</code>

Non si verificano altre sostituzioni di questo genere.

Le sequenze triple sono una novità dello standard ANSI.

### A.12.2 Congiunzione delle righe

La congiunzione di righe che terminano con il carattere barra inversa `\` avviene cancellando il carattere `\` e il carattere newline successivo. Ciò ha luogo prima della divisione in token.

### A.12.3 Definizione ed espansione delle macro

Una direttiva della forma

```
# define identificatore sequenza-token
```

fa sì che il preprocessore sostituisca le successive occorrenze dell'identificatore con la data sequenza di token; lo spazio bianco che precede e segue la sequenza di token viene scartato. Una seconda direttiva `#define` per lo stesso identificatore è errata a meno che la seconda sequenza di token sia identica alla prima, dove tutti gli spazi bianchi di separazione sono considerati equivalenti.

Una riga della forma

```
# define identificatore( lista-identificatorefac ) sequenza-token
```

dove non ci sono spazi tra il primo identificatore e la successiva parentesi tonda aperta, è la definizione di una macro con parametri dati dalla lista-identificatore. Come per la prima forma, è scartato lo spazio bianco intorno alla sequenza di token, sia in testa che in coda; la macro può essere ridefinita solo con una definizione identica nel numero e nel nome dei parametri, oltre che nella sequenza di token.

Una direttiva della forma

```
# undef identificatore
```

provoca l'annullamento della definizione dell'identificatore. Non è errato applicare `#undef` a un identificatore ignoto.

Quando una macro è stata definita nella seconda forma, costituiscono una chiamata della macro le occorrenze successive dell'identificatore della macro, seguite o meno dallo spazio bianco, da parentesi tonda aperta, da una sequenza di token separati da virgolette, e da parentesi tonda chiusa. Gli argomenti della chiamata sono le sequenze di token separate da virgolette; le virgolette tra virgolette o protette da parentesi annidate non separano gli argomenti. Nel corso della loro raccolta, agli argomenti non si applica l'espansione delle macro. Il numero di argomenti nella chiamata deve corrispondere al numero di parametri nella definizione. Dopo l'isolamento degli argomenti, viene eliminato lo spazio bianco che li precede e li segue. Quindi la sequenza di token che risulta da ciascun argomento è sostituita per ogni occorrenza (non fra apici) dell'identificatore del parametro corrispondente nella sequenza

di token di sostituzione della macro. A meno che il parametro nella sequenza di sostituzione sia preceduto da #, oppure preceduto o seguito da ##, i token degli argomenti sono esaminati per le chiamate della macro, ed espansi se necessario, appena prima dell'inserimento.

Due operatori speciali influenzano il processo di sostituzione. In primo luogo, se un'occorrenza di un parametro nella sequenza di token di sostituzione è immediatamente preceduta da #, le virgolette ("") precedono e seguono il parametro corrispondente, e quindi sia il # che l'identificatore del parametro vengono sostituiti dall'argomento tra virgolette. Un carattere \ è inserito prima di ogni carattere " o \ che compaia intorno o internamente a una costante stringa o a una costante carattere nell'argomento.

In secondo luogo, se la sequenza di token nella definizione di entrambi i tipi di macro contiene un operatore ##, subito dopo la sostituzione dei parametri, tutti i ## sono cancellati, insieme a tutti i caratteri di spazio bianco su entrambi i lati, così da concatenare i token adiacenti per formare un nuovo token. L'effetto è indefinito se si generano token non validi, o se il risultato dipende dall'ordine di elaborazione degli operatori ##. Inoltre, ## non può comparire all'inizio o alla fine di una sequenza di token di sostituzione.

In entrambi i generi di macro, la sequenza di token di sostituzione viene riesaminata a più riprese in cerca di altri identificatori definiti. Tuttavia, una volta che un certo identificatore sia stato rimpiazzato durante un'espansione, non sarà sostituito di nuovo se compare nei riesami successivi, e non subisce alcuna modifica.

Anche se il risultato finale dell'espansione di una macro dovesse cominciare con #, non verrebbe considerato una direttiva del preprocessore.

I dettagli del processo di espansione delle macro sono descritti con maggior esattezza nello standard ANSI che nella prima edizione. L'innovazione più rilevante è l'aggiunta degli operatori # e ##, che rendono ammissibile l'uso delle virgolette e della concatenazione. Alcune regole nuove, specialmente quelle riguardanti la concatenazione, sono bizzarre. (Si veda l'esempio seguente.)

Per esempio, questa funzionalità può essere usata per "costanti manifeste", come in

```
#define TABSIZE 100
int table[TABSIZE];
```

La definizione

```
#define ABSDIFF(a, b) ((a)>(b) ? (a)-(b) : (b)-(a))
```

dichiara una macro che restituiscia il valore assoluto della differenza tra i propri argomenti. Diversamente da una funzione che faccia la stessa cosa, gli argomenti e il valore restituito possono avere qualsiasi tipo aritmetico o anche essere puntatori. Inoltre, gli argomenti, che potrebbero generare effetti collaterali, sono valutati due volte, una per la condizione e una per produrre il valore.

Data la definizione

```
#define tempfile(dir) #dir "%s"
```

la chiamata della macro `tempfile(/usr/tmp)` genera

```
"/usr/tmp" "%s"
```

che subito dopo sarà concatenata in una stringa unica. In seguito a

```
#define cat(x, y) x ## y
```

la chiamata `cat(var, 123)` dà luogo a `var123`. Tuttavia, la chiamata `cat(cat(1,2),3)` è indefinita: la presenza di ## impedisce l'espansione degli argomenti della chiamata esterna. Infatti, la chiamata produce la stringa di token

```
cat ( 1 , 2 )3
```

e )3 (la concatenazione dell'ultimo token del primo argomento con il primo token del secondo) non è un token legale. Se si introduce una macro di secondo livello

```
#define xcat(x,y) cat(x,y)
```

si superano questi inconvenienti; `xcat(xcat(1, 2), 3)` genera in effetti 123, poiché l'espansione di xcat stessa non coinvolge l'operatore ##.

Analogamente, `ABSDIFF(ABSDIFF(a,b),c)` genera il risultato pienamente espanso che ci si aspetterebbe.

#### A.12.4 Inclusione di file

Una direttiva della forma

```
# include <filename>
```

viene interamente sostituita dai contenuti del file `filename`. I caratteri contenuti in `filename` non devono includere > o il carattere newline, e il risultato è indefinito qualora il nome ne contenga uno tra ", ', \ o /\*. Il file in questione è cercato in una serie di luoghi dipendenti dall'implementazione.

In maniera analoga, una direttiva della forma

```
# include "filename"
```

comincia a cercare il file richiesto da un punto dipendente dal file sorgente originale (i dettagli dipendono dall'implementazione), e se la ricerca fallisce, torna alla prima forma. Il risultato che si ottiene usando ', \ o /\* nel nome del file rimane indefinito, ma è permesso l'uso di >.

Infine, una direttiva della forma

```
# include sequence-token
```

che si discosti dalle espressioni precedenti è interpretata espandendo la sequenza di token come per il testo normale; il risultato deve corrispondere a una delle due forme precedenti con <...> o "...", che poi viene trattata come appena descritto.

I file #include possono essere annidati.

#### A.12.5 Compilazione condizionale

La compilazione condizionale si applica ad alcune parti di un programma, secondo la sintassi schematica che segue.

```

condizionale-preprocessore:
  riga-if testo parti-eliffac parte-elsefac # endif
  riga-if:
    # if espressione-costante
    # ifdef identificatore
    # ifndef identificatore
  parti-elif:
    riga-elif testo parti-eliffac
  riga-elif:
    # elif espressione-costante
  parte-else:
    riga-else testo
  riga-else:
    # else

```

Ciascuna direttiva (riga-if, riga-elif, riga-else e #endif) compare su una riga separata. Le espressioni costanti nelle direttive #if e le successive righe #elif sono valutate nell'ordine finché sia individuata un'espressione con un valore diverso da zero; il testo che segue una riga con un valore zero è scartato. Il testo successivo alla direttiva la cui condizione è soddisfatta è trattato normalmente. In questo caso “testo” significa qualunque materiale, righe del preprocessore incluse, che non faccia parte della struttura condizionale; può essere anche uno spazio vuoto. Non appena trovata una riga #if o #elif la cui condizione sia soddisfatta, ed elaborato il suo testo, vengono scartate le successive righe #elif e #else, nonché il loro testo. Se tutte le espressioni sono zero, e vi è un #else, il testo che segue l'#else è trattato normalmente. Il testo controllato da rami inattivi della struttura condizionale è ignorato fatta eccezione per la verifica dei condizionali annidati.

L'espressione costante in #if e #elif è soggetta alla ordinaria sostituzione delle macro. Inoltre, eventuali espressioni della forma

defined identificatore

o

defined ( identificatore )

sono sostituite, prima di analizzare le macro, da 1L se l'identificatore è definito nel preprocessore, e da 0L in caso contrario. Qualunque identificatore che rimanesse dopo l'espansione delle macro è sostituito da 0L. Infine, si presuppone che tutte le costanti intere rechino implicitamente il suffisso L, per cui tutta l'aritmetica è considerata long o unsigned long.<sup>(1)</sup>

1. N.d.R. Dall'errata corrigé pubblicata elettronicamente a cura degli Autori: “Il risultato dell'operatore `defined` non è sostituito letteralmente da 0L o 1L, ma semplicemente da 0 o 1; analogamente, i nomi indefiniti non sono sostituiti letteralmente da 0L, ma da 0. Tuttavia, l'espressione costante è ugualmente valutata come se queste e altre costanti presenti fossero di tipo long o unsigned long.”

L'espressione costante che ne risulta (§A.7.19) è soggetta a restrizioni: deve essere intera, e non può contenere sizeof, cast o costanti enumerative.

Le direttive

```
#ifdef identificatore
#ifndef identificatore
```

sono rispettivamente equivalenti a

```
#if defined identificatore
#ifndef identificatore
```

La direttiva #elif è nuova rispetto alla prima edizione, pur essendo stata disponibile in alcuni preprocessori. Anche l'operatore defined del preprocessore è nuovo.

## A.12.6 Controllo delle righe

A beneficio di altri preprocessori che generano programmi in C, una riga che abbia una delle forme

```
# line costante "nomefile"
# line costante
```

induce il compilatore a credere, a fini diagnostici, che il numero della riga successiva nel codice sia dato dalla costante intera decimale e che il file in ingresso corrente sia nominato dall'identificatore. Se il nome del file tra virgolette è assente, il nome conservato in memoria non cambia. Le macro della riga sono espanso prima che nomefile sia interpretato.

## A.12.7 Generazione di errori

Una riga del preprocessore della forma

```
# error sequenza-tokenfac
```

induce il processore a emettere un messaggio diagnostico che comprenda la sequenza di token.

## A.12.8 Pragma

Una direttiva della forma

```
# pragma sequenza-tokenfac
```

ordina al processore di attuare un'azione che dipende dall'implementazione. Un pragma non riconosciuto viene ignorato.

## A.12.9 Direttiva nulla

Una riga del preprocessore della forma

#

non ha alcun effetto.

### A.12.10 Nomi predefiniti

Numerosi identificatori sono predefiniti, e sono espansi al fine di fornire informazioni particolari. Essi, così come l'operatore `defined` del preprocessore, non possono essere ridefiniti, né si può loro applicare la direttiva `#undef`.

- `__LINE__` Una costante decimale che contiene il numero della riga di codice sorgente corrente.
- `__FILE__` Una costante stringa che contiene il nome del file in corso di compilazione.
- `__DATE__` Una costante stringa che contiene la data di compilazione, nella forma "Mmm gg aaaa".
- `__TIME__` Una costante stringa che contiene l'ora della compilazione, nella forma "hh:mm:ss".
- `__STDC__` La costante 1. Questo identificatore vale 1 esclusivamente nelle implementazioni conformi allo standard.

Le direttive `#error` e `#pragma` sono novità dello standard ANSI; le macro predefinite del preprocessore sono nuove, ma qualcuna di esse era disponibile in alcune implementazioni.

## A.13 Grammatica

Segue una ricapitolazione della grammatica del linguaggio presentata nel corso di questa appendice. I contenuti sono invariati, ma l'ordine di presentazione è diverso.

La grammatica prevede i simboli terminali non definiti *costante-intera*, *costante-carattere*, *costante-virgolamobile*, *identificatore*, *stringa* e *costante-enumerazione*; lo stile *typewriter* contraddistingue i simboli terminali dati letteralmente. La grammatica può essere trasformata meccanicamente in un formato accettabile da un generatore automatico di analizzatori sintattici. A tal fine, oltre ad aggiungere i simboli appropriati per segnalare le alternative nelle produzioni, è necessario riformulare i costrutti "uno fra". Inoltre, a seconda delle regole dello specifico generatore, può essere necessario sdoppiare le produzioni contrassegnate dal simbolo *fac* ("facoltativo") in una versione con il simbolo e in una senza. Se poi si rende *nome-typedef* un simbolo terminale eliminando la produzione *nome-typedef*:*identificatore*, la grammatica risultante sarà accettata dal generatore YACC. La grammatica presenta una sola situazione di conflitto, dovuta all'ambiguità relativa al costrutto *if-else*.

*unità-traduzione*:

- dichiarazione-esterna*
- unità-traduzione* *dichiarazione-esterna*

*dichiarazione-esterna*:

- definizione-funzione*
- dichiarazione*

*definizione-funzione*:

- specificatori-dichiarazione<sub>fac</sub>* *dichiaratore* *lista-dichiarazione<sub>fac</sub>* *istruzione-composta*

*dichiarazione*:

- specificatori-dichiarazione* *lista-iniz-dichiaratore<sub>fac</sub>* ;

*lista-dichiarazione*:

- dichiarazione*
- lista-dichiarazione* *dichiarazione*

*specificatori-dichiarazione*:

- specificatore-classe-memoria* *specificatori-dichiarazione<sub>fac</sub>*
- specificatore-tipo* *specificatori-dichiarazione<sub>fac</sub>*
- qualificatore-tipo* *specificatori-dichiarazione<sub>fac</sub>*

*specificatore-classe-memoria*: uno fra

- auto*
- register*
- static*
- extern*
- typedef*

*specificatore-tipo*: uno fra

- void*
- char*
- short*
- int*
- long*
- float*
- double*
- signed*
- unsigned*

- specificatore-strutt-o-unione*
- specificatore-enum*
- nome-typedef*

*qualificatore-tipo*: uno fra

- const*
- volatile*

*specificatore-strutt-o-unione*:

- strutt-o-unione* *identificatore<sub>fac</sub>* { *lista-dichiarazione-strutt* }
- strutt-o-unione* *identificatore*

*strutt-o-unione*: uno fra

- struct*
- union*

*lista-dichiarazione-strutt*:

- dichiarazione-strutt*
- lista-dichiarazione-strutt* *dichiarazione-strutt*

*lista-iniz-dichiaratore*:

- iniz-dichiaratore*
- lista-iniz-dichiaratore* , *iniz-dichiaratore*

*iniz-dichiaratore*:

- dichiaratore*
- dichiaratore* = *inizializzatore*

*dichiarazione-strutt*:

- lista-specificatore-qualificatore* *lista-dichiaratore-strutt* ;

*lista-specificatore-qualificatore*:

- specificatore-tipo* *lista-specificatore-qualificatore<sub>fac</sub>*
- qualificatore-tipo* *lista-specificatore-qualificatore<sub>fac</sub>*

*lista-dichiaratore-strutt:*

- dichiaratore-strutt*
- lista-dichiaratore-strutt , dichiaratore strutt*

*dichiaratore-strutt:*

- dichiaratore*
- dichiaratore<sub>fac</sub> : espressione-costante*

*specificatore-enum:*

- enum identificatore<sub>fac</sub> { lista-enumeratore }*
- enum identificatore*

*lista-enumeratore:*

- enumeratore*
- lista-enumeratore , enumeratore*

*enumeratore:*

- identificatore*
- identificatore = espressione-costante*

*dichiaratore:*

- puntatore<sub>fac</sub> dichiaratore-diretto*

*dichiaratore-diretto:*

- identificatore*
- ( dichiaratore )*
- dichiaratore-diretto [ espressione-costante<sub>fac</sub> ]*
- dichiaratore-diretto ( lista-tipo-parametro )*
- dichiaratore-diretto ( lista-identificatore<sub>fac</sub> )*

*puntatore:*

- \* lista-qualificatore-tipo<sub>fac</sub>*
- \* lista-qualificatore-tipo<sub>fac</sub> puntatore*

*lista-qualificatore-tipo:*

- qualificatore-tipo*
- lista-qualificatore-tipo qualificatore-tipo*

*lista-tipo-parametro:*

- lista-parametro*
- lista-parametro , ...*

*lista-parametro:*

- dichiarazione-parametro*
- lista-parametro , dichiarazione-parametro*

*dichiarazione-parametro:*

- specificatori-dichiarazione dichiaratore*
- specificatori-dichiarazione dichiaratore-astratto<sub>fac</sub>*

*lista-identificatore:*

- identificatore*
- lista-identificatore , identificatore*

*inizializzatore:*

- espressione-assegnamento*
- { lista-inizializzatore }*
- { lista-inizializzatore , }*

*lista-inizializzatore:*

- inizializzatore*
- lista-inizializzatore , inizializzatore*

*nome-tipo:*

- lista-specificatore-qualificatore dichiaratore-astratto<sub>fac</sub>*

*dichiaratore-astratto:*

- puntatore*
- puntatore<sub>fac</sub> dichiaratore-diretto-astratto*

*dichiaratore-diretto-astratto:*

- ( dichiaratore-astratto )*
- dichiaratore-diretto-astratto<sub>fac</sub> [ espressione-costante<sub>fac</sub> ]*
- dichiaratore-diretto-astratto<sub>fac</sub> ( lista-tipo-parametro<sub>fac</sub> )*

*nome-typedef:*

- identificatore*

*istruzione:*

- istruzione-etichettata*
- istruzione-espressione*
- istruzione-composta*
- istruzione-scelta*
- istruzione-iterazione*
- istruzione-salto*

*istruzione-etichettata:*

- identificatore : istruzione*
- case espressione-costante : istruzione*
- default : istruzione*

*istruzione-espressione:*

espressione<sub>fac</sub> ;

*istruzione-composta:*

{ lista-dichiarazione<sub>fac</sub> lista-istruzione<sub>fac</sub> }

*lista-istruzione:*

istruzione

lista-istruzione istruzione

*istruzione-scelta:*

if ( espressione ) istruzione

if ( espressione ) istruzione else istruzione

switch ( espressione ) istruzione

*istruzione-iterazione:*

while ( espressione ) istruzione

do istruzione while ( espressione ) ;

for ( espressione<sub>fac</sub> ; espressione<sub>fac</sub> ; espressione<sub>fac</sub> ) istruzione

*istruzione-salto:*

goto identificatore ;

continue ;

break ;

return espressione<sub>fac</sub> ;

*espressione:*

espressione-assegnamento

espressione , espressione-assegnamento

*espressione-assegnamento:*

espressione-condizionale

espressione-unaria operatore-assegnamento espressione-assegnamento

*operatore-assegnamento:* uno fra

= \* = / = % = + = - = << = >> = & = ^ = | =

*espressione-condizionale:*

espressione-OR-logico

espressione-OR-logico ? espressione : espressione-condizionale

*espressione-costante:*

espressione-condizionale

*espressione-OR-logico:*

espressione-AND-logico

espressione-OR-logico || espressione-AND-logico

*espressione-AND-logico:*

espressione-OR-inclusivo

espressione-AND-logico && espressione-OR-inclusivo

*espressione-OR-inclusivo:*

espressione-OR-esclusivo

espressione-OR-inclusivo | espressione-OR-esclusivo

*espressione-OR-esclusivo:*

espressione-AND

espressione-OR-esclusivo ^ espressione-AND

*espressione-AND:*

espressione-uguaglianza

espressione-AND & espressione-uguaglianza

*espressione-uguaglianza:*

espressione-relazionale

espressione-uguaglianza == espressione-relazionale

espressione-uguaglianza != espressione-relazionale

*espressione-relazionale:*

espressione-scorrimento

espressione-relazionale < espressione-scorrimento

espressione-relazionale > espressione-scorrimento

espressione-relazionale <= espressione-scorrimento

espressione-relazionale >= espressione-scorrimento

*espressione-scorrimento:*

espressione-additiva

espressione-scorrimento << espressione-additiva

espressione-scorrimento >> espressione-additiva

*espressione-additiva:*

espressione-moltiplicativa

espressione-additiva + espressione-moltiplicativa

espressione-additiva - espressione-moltiplicativa

*espressione-moltiplicativa:*

espressione-cast

espressione-moltiplicativa \* espressione-cast

espressione-moltiplicativa / espressione-cast

espressione-moltiplicativa % espressione-cast

*espressione-cast:*

```
espressione-unaria
( nome-tipo ) espressione-cast
```

*espressione-unaria:*

```
espressione-postfissa
++ espressione-unaria
-- espressione-unaria
operatore-unario espressione-cast
sizeof espressione-unaria
sizeof ( nome-tipo )
```

*operatore-unario:* uno fra

```
& * + - ~ !
```

*espressione-postfissa:*

```
espressione-primaria
espressione-postfissa [ espressione ]
espressione-postfissa ( lista-argomenti-espressionefac )
espressione-postfissa . identificatore
espressione-postfissa -> identificatore
espressione-postfissa ++
espressione-postfissa --
```

*espressione-primaria:*

```
identificatore
costante
stringa
( espressione )
```

*lista-argomenti-espressione:*

```
espressione-assegnamento
lista-argomenti-espressione , espressione-assegnamento
```

*costante:*

```
costante-intera
costante-carattere
costante-virgolamobile
costante-enumerazione
```

La seguente grammatica per il preprocessore riassume la struttura delle righe di controllo, ma non è adatta all'analisi sintattica automatica. La grammatica comprende il simbolo *testo*, che rappresenta l'ordinario testo del programma, le righe di controllo del preprocessore non condizionali o i costrutti condizionali completi del preprocessore.

*riga-controllo:*

```
# define identificatore sequenza-token
# define identificatore( identificatore , ... , identificatore ) sequenza-token
# undef identificatore
# include <nomefile>
# include "nomefile"
# include sequenza-token
# line costante "nomefile"
# line costante
# error sequenza-tokenfac
# pragma sequenza-tokenfac
#
condizionale-preprocessore
```

*condizionale-preprocessore:*

```
riga-if testo parti-eliffac parte-elsefac # endif
```

*riga-if:*

```
# if espressione-costante
# ifdef identificatore
# ifndef identificatore
```

*parti-elif:*

```
riga-elif testo parti-eliffac
```

*riga-elif:*

```
# elif espressione-costante
```

*parte-else:*

```
riga-else testo
```

*riga-else:*

```
# else
```

# B

## Libreria standard

**Q**UESTA APPENDICE RIASSUME LA LIBRERIA DEFINITA DALLO STANDARD ANSI. Essa non fa propriamente parte del linguaggio, ma gli ambienti che supportano il C in versione standard devono anche mettere a disposizione le dichiarazioni delle funzioni e le definizioni dei tipi e delle macro di questa libreria. Abbiamo omesso alcune funzioni di modesta utilità o facilmente ricostruibili a partire da quelle descritte, non trattiamo i caratteri estesi, e non ci occupiamo delle questioni relative a lingue, nazionalità e culture diverse.

Funzioni, tipi e macro della libreria standard sono dichiarati nelle *intestazioni* standard:

<assert.h>	<float.h>	<math.h>	<stdarg.h>	<stdlib.h>
<ctype.h>	<limits.h>	<setjmp.h>	<stddef.h> <sup>(1)</sup>	<string.h>
<errno.h>	<locale.h>	<signal.h>	<stdio.h>	<time.h>

Le intestazioni si allegano al sorgente in questo modo:

```
#include <intestazione>
```

---

1. *N.d.R.* Come segnalato dagli Autori nell'errata corrige pubblicata elettronicamente, in questa appendice manca il paragrafo dedicato all'intestazione `<stddef.h>`.

Le intestazioni possono essere incluse in qualunque ordine e un numero di volte impreciso, ma sempre al di fuori di una definizione o dichiarazione esterna, e prima dell'uso di qualunque cosa dichiarata dall'intestazione stessa. Le intestazioni non devono necessariamente essere incluse nel file sorgente.

L'uso degli identificatori esterni che iniziano con underscore (\_) è riservato alla libreria, così come quello di tutti gli identificatori che iniziano con doppio underscore (\_\_) o con un singolo underscore seguito da lettera maiuscola.

## B.1 Ingresso e uscita dei dati: <stdio.h>

Le funzioni, i tipi e le macro definite in <stdio.h> per agevolare l'ingresso e l'uscita dei dati rappresentano quasi un terzo dell'intera libreria.

Un *flusso* è una fonte o una destinazione di dati, e può essere associato a un disco o ad altri dispositivi periferici. La libreria prevede flussi di testo e binari, anche se per alcuni sistemi, in particolare per UNIX, non vi è alcuna differenza. Un flusso di testo è una successione di righe, ciascuna delle quali è costituita da zero o più caratteri e termina con '\n'. Un dato ambiente potrà aver bisogno di convertire un flusso di testo in un'altro formato, e viceversa; per esempio, '\n' può dover essere interpretato come un ritorno del carrello e un fine riga (carriage return e linefeed). Un flusso binario è una successione di byte, senza alcuna struttura, che rappresenta dati in formato interno, con la proprietà che sullo stesso sistema la successione rimanga invariata dopo scrittura seguita da lettura.

L'associazione fra un flusso e un dispositivo periferico si instaura *aprendo* il flusso, e si rimuove *chiudendo* il flusso. L'apertura di un file restituisce un puntatore a un oggetto di tipo FILE, che registra le informazioni necessarie per controllare il flusso. Useremo "puntatore a file" e "flusso" come sinonimi, laddove non vi sia pericolo di ambiguità.

Nel momento in cui inizia l'esecuzione di un programma, i tre flussi stdin, stdout e stderr sono già aperti.

### B.1.1 Operazioni sui file

Le funzioni seguenti eseguono operazioni sui file. Il tipo size\_t è il tipo intero senza segno restituito dall'operatore sizeof.

`FILE *fopen(const char *filename, const char *mode)`

Apre il file filename e restituisce un flusso, o NULL se l'apertura non riesce. I valori ammessi per la modalità d'apertura (mode) includono i seguenti.

- "r" Apre un file di testo per la lettura ("read").
- "w" Crea un file di testo per la scrittura ("write"); elimina eventuali contenuti precedenti del file.
- "a" Aggiunge in coda ("append"); apre o crea un file di testo per la scrittura alla fine del file.
- "r+" Apre un file di testo per l'aggiornamento, cioè lettura e scrittura.
- "w+" Crea un file di testo per l'aggiornamento; elimina eventuali contenuti precedenti del file.

"a+" Aggiunge in coda; apre o crea un file di testo per l'aggiornamento, dove la scrittura avviene a partire dalla fine del file.

La modalità di aggiornamento permette la lettura e la scrittura sullo stesso file; fra un'operazione di lettura e una di scrittura, o viceversa, è necessario chiamare fflush o una funzione per il posizionamento all'interno del file. Una modalità che includa b dopo la lettera iniziale, come in "rb" o "w+b", denota un file binario. I nomi dei file non possono avere più di FILENAME\_MAX caratteri, e al massimo FOPEN\_MAX file possono essere aperti in contemporanea.

`FILE *freopen(const char *filename, const char *mode, FILE *stream)`

Apre il file con la modalità specificata e gli associa il flusso stream. Restituisce stream, o NULL in caso d'errore. È solitamente usata per modificare i file associati a stdin, stderr e stdout.

`int fflush(FILE *stream)`

Applicata a un flusso in uscita, fflush forza l'effettiva scrittura di eventuali dati presenti nel buffer; applicata a un flusso in ingresso, l'effetto è indefinito. Restituisce EOF per un errore durante la scrittura, e zero altrimenti. La chiamata fflush(NULL) applica fflush a tutti i flussi aperti in uscita.

`int fclose(FILE *stream)`

Questa funzione svuota eventualmente il buffer del flusso stream (come cioè se si invocasse fflush(stream)), elimina eventuali dati in ingresso non ancora letti e residenti nel buffer, libera gli eventuali buffer allocati automaticamente, e chiude il flusso. Restituisce EOF in caso d'errore, e zero altrimenti.

`int remove(const char *filename)`

Rimuove il file argomento, in modo da impedire la riuscita di successivi tentativi di aprirlo. Restituisce un valore non nullo se non si riesce a rimuovere il file.

`int rename(const char *oldname, const char *newname)`

Modifica il nome di un file. Restituisce un valore non nullo se non si riesce a modificare il nome del file.

`FILE *tmpfile(void)`

Crea un file temporaneo con modalità "wb+" che sarà automaticamente rimosso alla chiusura, o quando il programma termina normalmente. Restituisce un flusso, o NULL se non si riesce a creare il file.

`char *tmpnam(char s[L_tmpnam])`

La chiamata tmpnam(NULL) crea una stringa che non coincide con il nome di alcun file esistente, e restituisce un puntatore a un vettore interno statico. La chiamata tmpnam(s) memorizza la stringa in questione in s, oltre a restituirla come valore della funzione; s deve poter contenere almeno L\_tmpnam caratteri. La funzione genera un nome diverso a ogni chiamata, garantendo al massimo TMP\_MAX nomi diversi durante l'esecuzione del programma. Si noti che la funzione genera un nome, non un file.

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size)
```

Gestisce i buffer associati a un flusso; deve essere chiamata prima della lettura, della scrittura, o di qualsivoglia operazione. La modalità (mode) `_IOFBF` (IO Full Buffering) richiede l'uso dei buffer a pieno regime, `_IOLBF` (Line Buffering) l'uso di buffer riga per riga per i file di testo, `_IONBF` (No Buffering) l'assenza di buffer. Se `buf` non è `NULL`, sarà impiegato come buffer; altrimenti, sarà allocato un buffer. Il parametro `size` determina la dimensione del buffer. Restituisce un valore non nullo in presenza di eventuali errori.

```
void setbuf(FILE *stream, char *buf)
```

Se `buf` è `NULL`, elimina l'uso dei buffer per il flusso `stream`; altrimenti, la sua invocazione è equivalente a `(void) setvbuf(stream, buf, _IOFBF, BUFSIZ)`.

### B.1.2 Formattazione dei dati in uscita

Le funzioni della famiglia `printf` permettono la formattazione dei dati in uscita.

```
int fprintf(FILE *stream, const char *format, ...)
```

Converte e scrive dati in uscita sul flusso `stream`, in accordo alle specifiche dettate da `format`. Restituisce il numero di caratteri scritti, o un numero negativo in caso d'errore.

La stringa `format` contiene due tipi di oggetti: caratteri ordinari, che sono copiati sul flusso in uscita, e specifiche di conversione, ognuna delle quali causa la conversione e la visualizzazione dell'argomento corrispondente di `fprintf` nell'ellissi. Le specifiche di conversione iniziano con `%` e terminano con un carattere che detta il tipo di conversione richiesta. Fra il `%` e il carattere di conversione ci possono essere, nell'ordine:

- Opzioni (flag), in qualunque ordine, che modifichino le specifiche:
  - , che specifica l'allineamento a sinistra, nel suo campo, dell'argomento convertito.
  - + , che specifica che il numero sarà sempre visualizzato con un segno.
  - spazio*: se il primo carattere non è un segno, si premetterà uno spazio.
  - 0: per le conversioni numeriche, richiede l'inserzione di zeri in testa fino al raggiungimento dell'ampiezza del campo.
  - #, che specifica un formato di output alternativo. Per o, la prima cifra sarà zero. Per x o X, il prefisso `0x` o `0X` sarà aggiunto a un risultato non nullo. Per e, E, f, g, G, il dato avrà sempre il punto decimale; per g e G, non saranno rimossi gli zeri in coda.
- Un numero che specifichi l'ampiezza minima del campo. L'argomento convertito sarà visualizzato in un campo che abbia come minimo tale dimensione, o sia più ampio se necessario. Se l'argomento convertito ha meno caratteri dell'ampiezza minima del campo, saranno aggiunti in testa dei caratteri (o in coda, se è richiesto l'allineamento a sinistra) fino a raggiungere l'ampiezza richiesta; di solito il carattere impiegato è lo spazio, ma è 0 se si è richiesta esplicitamente questa opzione.
- Un punto, per separare l'ampiezza del campo dal grado di precisione.
- Un numero, il grado di precisione, che specifichi: per le stringhe, il massimo numero di caratteri da scrivere; per le conversioni e, E o f, il numero di cifre da visualizzare dopo il punto decimale; per g o G, il numero di cifre significative; per gli interi, il numero

minimo di cifre da scrivere (con l'aggiunta di zeri in testa per raggiungere l'ampiezza richiesta).

- Un modificatore della lunghezza h, l (lettera elle) o L: "h" indica che l'argomento corrispondente è da scrivere come `short` o `unsigned short`; "l" che l'argomento corrispondente è di tipo `long` o `unsigned long`; "L" che l'argomento è di tipo `long double`.

Per l'ampiezza, il grado di precisione o entrambi si può anche specificare \*, nel qual caso il valore è calcolato convertendo il successivo o i successivi argomenti, che devono essere di tipo `int`.

I caratteri di conversione e il loro significato sono riassunti nella Tabella B.1. Se il carattere successivo a % non dà luogo a una specifica di conversione ammessa, il comportamento del programma è indefinito.

```
int printf(const char *format, ...)
printf(...) equivale a fprintf(stdout, ...).
```

**Tabella B.1** Conversioni di `printf`.

CARATTERE	TIPO DELL'ARGOMENTO; SCRITTO COME
d, i	int; notazione decimale con segno.
o	unsigned int; notazione ottale priva di segno (senza lo zero in testa).
x, X	unsigned int; notazione esadecimale priva di segno (senza <code>0x</code> o <code>0X</code> in testa), dove abcdef (per <code>0x</code> ) o ABCDEF (per <code>0X</code> ) stanno per 10, ..., 15.
u	unsigned int; notazione decimale priva di segno.
c	int; carattere singolo, dopo la conversione in <code>unsigned char</code> .
s	char *; stampa i caratteri della stringa fino a raggiungere '\0', o a esaurire il grado di precisione specificato.
f	double; notazione decimale della forma [-]iii.ddd, dove iii è la parte intera, ddd sono le cifre dopo il punto decimale, in numero dato dal grado di precisione specificato (di default è pari a 6); omette il punto decimale se il grado di precisione è zero.
e, E	double; notazione decimale della forma [-]i.dddddde±xx o [-]i.dddddE±xx, dove xx è l'esponente, e il numero di d è dato dal grado di precisione specificato (di default è pari a 6); omette il punto decimale se il grado di precisione è zero.
g, G	double; usa %e o %E se l'esponente è minore di -4 o maggiore o uguale al grado di precisione; altrimenti usa %f. Omette gli zeri in coda, e il punto decimale in coda.
p	void *; puntatore (rappresentazione dipendente dall'implementazione).
n	int *; il numero di caratteri scritti finora da questa chiamata a <code>printf</code> è salvato nell'argomento. Non ha luogo alcuna conversione.
%	non ha luogo alcuna conversione; visualizza un %.

```
int sprintf(char *s, const char *format, ...)
```

Come printf, eccetto che i dati sono scritti sulla stringa s e terminano con '\0'. La stringa deve essere grande a sufficienza per contenere il risultato. Il valore restituito, che è pari al numero di caratteri scritti, non include '\0'.

```
int vprintf(const char *format, va_list arg)
int vfprintf(FILE *stream, const char *format, va_list arg)
int vsprintf(char *s, const char *format, va_list arg)
```

Queste funzioni sono equivalenti alle corrispondenti funzioni printf, salvo che la lista di argomenti di lunghezza variabile è sostituita da arg, che è stato inizializzato tramite la macro va\_start e, potenzialmente, da chiamate a va\_arg. Si veda l'analisi di <stdarg.h> nel Paragrafo B.7.

### B.1.3 Formattazione dei dati in ingresso

Le funzioni della famiglia scanf gestiscono la formattazione dei dati in ingresso.

```
int fscanf(FILE *stream, const char *format, ...)
```

Legge dal flusso stream sotto il controllo di format, e assegna i valori convertiti attraverso gli argomenti successivi, *ognuno dei quali deve essere un puntatore*. Termina quando format è stata completamente esaminata. Restituisce EOF se incontra la fine del file, o incorre in un errore prima di avere eseguito una qualsiasi conversione; altrimenti, restituisce il numero di argomenti convertiti e assegnati con successo.

Di norma, la stringa format contiene specifiche di conversione che servono a guidare l'interpretazione dei dati in ingresso. Essa può contenere:

- Spazi o caratteri di tabulazione, che sono ignorati.<sup>(2)</sup>
- Caratteri ordinari (diversi da %), che devono collimare esattamente con i corrispondenti caratteri del flusso in ingresso.
- Le specifiche di conversione, che consistono del carattere %, di un carattere facoltativo \* per la soppressione dell'assegnamento, di un numero facoltativo che precisa l'ampiezza massima del campo, eventualmente di una h, l o L che indicano la dimensione dello spazio riservato alla memorizzazione del dato in ingresso, e infine di un carattere di conversione.

Una specifica di conversione presiede alla conversione del successivo campo in ingresso. Normalmente il risultato è collocato nella variabile a cui punta l'argomento corrispondente. Se la soppressione dell'assegnamento è segnalata dal carattere \*, come in %\*s, tuttavia, si ignora semplicemente il campo in ingresso corrispondente; non viene fatto alcun assegnamento. Un campo in ingresso è definito come una stringa di caratteri che non siano spazi, e si estende o fino allo spazio successivo o fin dove arriva l'ampiezza del cam-

2. N.d.R. Nella errata corrigé si legge: "Le funzioni scanf non ignorano gli spazi nella stringa di composizione; se essa contiene degli spazi in un dato punto, tutti gli spazi nel corrispondente input vengono saltati".

**Tabella B.2** Conversioni di scanf.

CARATTERE	DATI IN INGRESSO; TIPO DELL'ARGOMENTO
d	intero in notazione decimale; int *
i	intero; int *. L'intero può essere in notazione ottale (segnalata da uno zero in testa) o esadecimale (segnalata da 0x o 0X in testa).
o	intero in notazione ottale privo di segno (con o senza lo zero in testa); unsigned int *
u	intero in notazione decimale privo di segno; unsigned int *
x	intero in notazione esadecimale (con o senza 0x o 0X in testa); unsigned int *
c	caratteri; char *. I successivi caratteri in ingresso, in numero indicato dall'ampiezza del campo (di default è pari a 1), sono memorizzati nel vettore indicato. Non è aggiunto '\0'. Anche gli spazi contano come caratteri: per leggere il carattere successivo che non sia uno spazio, si usi %1s.
s	stringa di caratteri che non siano spazi (non fra virgolette); char *, che punta a un vettore di caratteri di lunghezza sufficiente a contenere la stringa, più il carattere aggiunto di terminazione '\0'.
e, f, g	numero con virgola mobile; float *. Il formato prevede un segno facoltativo, una stringa di numeri che può contenere un punto decimale facoltativo, un campo facoltativo per l'esponente che contenga E o e seguito da un intero con segno facoltativo.
p	valore di un puntatore, nel formato scritto da printf("%p"); void *
n	scrive nell'argomento il numero di caratteri letti finora da questa chiamata; int *. Non legge dati in ingresso. Non incrementa il contatore delle conversioni eseguite.
[...] -	legge la stringa non vuota più lunga in ingresso che contenga caratteri elencati fra parentesi quadre; char *. Aggiunge '\0'. La forma [...] aggiunge ] alla lista di caratteri.
[^...]	legge la stringa non vuota più lunga in ingresso che non contenga caratteri elencati fra parentesi quadre; char *. Aggiunge '\0'. La forma [^...] aggiunge ] alla lista di caratteri.
%	occorrenza letterale di % in ingresso; non ha luogo alcun assegnamento.

po, se specificata. Questo significa che scanf, leggendo, potrà passare da una riga all'altra, perché i caratteri newline sono considerati spazi. (Appartengono a questa categoria lo spazio bianco ordinario, il carattere di tabulazione, il carattere newline, il ritorno del carrello, il carattere di tabulazione verticale e il salto pagina.)

Il carattere di conversione stabilisce l'interpretazione da dare al campo in ingresso. L'argomento corrispondente deve essere un puntatore. Le specifiche di conversione sono illustrate nella Tabella B.2.

I caratteri di conversione d, i, n, o, u e x possono essere preceduti da h se l'argomento è un puntatore al tipo short anziché int, o da l (lettera elle) se l'argomento è un puntatore al tipo long. I caratteri di conversione e, f e g possono essere preceduti da l se l'argomento corrispondente nell'elenco è un puntatore a double e non a float, e da L se esso è un puntatore a long double.

```
int scanf(const char *format, ...)
```

La funzione `scanf(...)` è identica a `fscanf(stdin,...)`.

```
int sscanf(const char *s, const char *format, ...)
```

La funzione `sscanf(s,...)` è equivalente a `scanf(...)`, eccetto che i caratteri in ingresso sono presi dalla stringa `s`.

#### B.1.4 Funzioni di input/output dei caratteri

```
int fgetc(FILE *stream)
```

Restituisce il successivo carattere del flusso `stream` come `unsigned char` (convertito in `int`), o `EOF` se incorre nella fine del file o in un errore.

```
char *fgets(char *s, int n, FILE *stream)
```

Legge al massimo `i` successivi `n-1` caratteri, copiandoli nel vettore `s`, fermandosi se trova un carattere newline, che è incluso nel vettore, che è terminato da '`\0`'. Restituisce `s`, o `NULL` se incontra la fine del file o incorre in un errore.

```
int fputc(int c, FILE *stream)
```

Scrive il carattere `c` (convertito in `unsigned char`) sul flusso `stream`. Restituisce il carattere scritto, o `EOF` se incorre in un errore.

```
int fputs(const char *s, FILE *stream)
```

Scrive la stringa `s` (che non deve necessariamente contenere '`\0`') sul flusso `stream`; restituisce valore non negativo, o `EOF` se incorre in un errore.

```
int getc(FILE *stream)
```

È equivalente a `fgetc`, eccetto che, nel caso sia una macro, potrebbe valutare `stream` più di una volta.

```
int getchar(void)
```

Equivale a `getc(stdin)`.

```
char *gets(char *s)
```

Legge la riga in ingresso successiva, e la memorizza nel vettore `s`; sostituisce il carattere newline con '`\0`'. Restituisce `s`, o `NULL` se incontra la fine del file o incorre in un errore.

```
int putc(int c, FILE *stream)
```

Equivale a `fputc`, eccetto che, nel caso sia una macro, potrebbe valutare `stream` più di una volta.

```
int putchar(int c)
```

Equivale a `putc(c, stdout)`.

```
int puts(const char *s)
```

Scrive la stringa `s`, seguita da un carattere newline, su `stdout`. Restituisce `EOF` se incorre in un errore, e un valore non negativo altrimenti.

```
int ungetc(int c, FILE *stream)
```

Riconsegna `c` (convertito in `unsigned char`) al flusso `stream`, da dove sarà restituito alla successiva operazione di lettura. Garantisce la riconsegna di un solo carattere per flusso; non si può riconsegnare `EOF`. Restituisce il carattere riconsegnato al flusso, o `EOF` se incorre in un errore.

#### B.1.5 Funzioni di input/output diretto

```
size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)
```

Legge dal flusso `stream` al massimo `nobj` oggetti di dimensione `size`, e li memorizza nel vettore `ptr`; restituisce il numero di oggetti letti, che può risultare inferiore al numero richiesto. Per determinare l'esito dell'operazione è necessario usare `feof` e `ferror`.

```
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)
```

Scrive `nobj` oggetti di dimensione `size` dal vettore `ptr` sul flusso `stream`. Restituisce il numero di oggetti scritti, che risulta minore di `nobj` se si incorre in un errore.

#### B.1.6 Funzioni per il posizionamento all'interno di file

```
int fseek(FILE *stream, long offset, int origin)
```

Imposta la posizione corrente per il file `stream`; un'operazione successiva di lettura o scrittura accederà ai dati a partire da quella posizione. Se il file è binario, la posizione corrisponde a `offset` caratteri a partire da `origin`, che può essere `SEEK_SET` (inizio del file), `SEEK_CUR` (posizione corrente) o `SEEK_END` (fine del file). Se il file è di testo, `offset` deve valere zero, o deve essere un valore restituito da `ftell` (nel qual caso `origin` deve valere `SEEK_SET`). Restituisce un valore non nullo se incorre in un errore.

```
long ftell(FILE *stream)
```

Restituisce la posizione corrente del file `stream`, o `-1L` in caso d'errore.

```
void rewind(FILE *stream)
```

La chiamata `rewind(fp)` equivale a `fseek(fp,0L,SEEK_SET); clearerr(fp)`.

```
int fgetpos(FILE *stream, fpos_t *ptr)
```

Registra la posizione corrente per `stream` in `*ptr`, a beneficio di chiamate successive a `fsetpos`. Il tipo `fpos_t` è adatto alla custodia di tali valori. Restituisce un valore diverso da zero in caso d'errore.

```
int fsetpos(FILE *stream, const fpos_t *ptr)
```

Imposta la posizione corrente per `stream` al punto registrato da `fgetpos` in `*ptr`. Restituisce un valore diverso da zero in caso d'errore.

### B.1.7 Funzioni per la gestione degli errori

Molte funzioni della libreria, nel caso incorrano in errori o incontrino la fine del file, attivano indicatori dello stato delle operazioni. Gli indicatori possono anche essere attivati ed esaminati esplicitamente. In più, l'espressione intera `errno`, dichiarata in `<errno.h>`, può contenere un numero dell'errore che fornisca informazioni aggiuntive sull'errore verificatosi più di recente.

```
void clearerr(FILE *stream)
    Disattiva gli indicatori d'errore e fine del file relativi a stream.
```

```
int feof(FILE *stream)
    Restituisce un valore non nullo se è attivo l'indicatore della fine del file relativo a stream.
```

```
int ferror(FILE *stream)
    Restituisce un valore non nullo se è attivo l'indicatore d'errore relativo a stream.
```

```
void perror(const char *s)
    Visualizza s e un messaggio d'errore definito dall'implementazione corrispondente all'intero in errno, come se si invocasse:
        fprintf(stderr, "%s: %s\n", s, "messaggio d'errore")
    Si veda strerror nel Paragrafo B.3.
```

## B.2 Classi di caratteri: <ctype.h>

L'intestazione `<ctype.h>` definisce funzioni di verifica dei caratteri. L'argomento di tutte le funzioni è del tipo `int`, il cui valore deve essere EOF oppure rappresentabile come `unsigned char`; il valore restituito è intero. Le funzioni restituiscono un valore diverso da zero (vero) se l'argomento `c` soddisfa la condizione descritta nell'elenco seguente, e zero altrimenti.

<code>isalnum(c)</code>	<code>isalpha(c) o isdigit(c)</code> è vero
<code>isalpha(c)</code>	<code>isupper(c) o islower(c)</code> è vero
<code>iscntrl(c)</code>	Carattere di controllo
<code>isdigit(c)</code>	Cifra decimale
<code>isgraph(c)</code>	Carattere visualizzabile, eccetto gli spazi
<code>islower(c)</code>	Minuscolo
<code>isprint(c)</code>	Carattere visualizzabile, inclusi gli spazi
<code>ispunct(c)</code>	Carattere visualizzabile, eccetto spazi, lettere e cifre
<code>isspace(c)</code>	Spazio, salto pagina, newline, ritorno del carrello, tabulazione, tabulazione verticale
<code>isupper(c)</code>	Maiuscolo
<code>isxdigit(c)</code>	Cifra esadecimale

Nel set di caratteri ASCII a sette bit, i caratteri visualizzabili vanno da `0x20` (' ') a `0x7E` ('~'); i caratteri di controllo vanno da `0` (NUL) a `0x1F` (US), e includono anche `0x7F` (DEL).

Inoltre, esistono due funzioni per la conversione di maiuscole e minuscole:

```
int tolower(int c)    Converte c in minuscola
int toupper(int c)   Converte c in maiuscola
```

Se `c` è maiuscola, `tolower(c)` restituisce la lettera minuscola corrispondente; altrimenti restituisce `c`. Se `c` è minuscola, `toupper(c)` restituisce la lettera maiuscola corrispondente; altrimenti restituisce `c`.

## B.3 Funzioni per la gestione delle stringhe: <string.h>

L'intestazione `<string.h>` contiene due gruppi di funzioni relative alle stringhe: quelle del primo gruppo hanno nomi che iniziano per `str`, quelle del secondo per `mem`. A eccezione di `memmove`, il comportamento del codice è indefinito se si tenta di copiare informazioni fra due oggetti parzialmente sovrapposti. Le funzioni di confronto trattano i loro argomenti alla stregua di vettori del tipo `unsigned char`.

Nell'elenco seguente, le variabili `s` e `t` sono del tipo `char *`, `cs` e `ct` del tipo `const char *`, `n` del tipo `size_t`, e `c` del tipo `int` convertito in `char`.

<code>char *strcpy(s,ct)</code>	Copia la stringa <code>ct</code> nella stringa <code>s</code> , incluso '\0'; restituisce <code>s</code> .
<code>char *strncpy(s,ct,n)</code>	Copia al massimo <code>n</code> caratteri della stringa <code>ct</code> in <code>s</code> ; restituisce <code>s</code> . Riempie con caratteri '\0' se <code>ct</code> ha meno di <code>n</code> caratteri.
<code>char *strcat(s,ct)</code>	Concatena la stringa <code>ct</code> alla fine della stringa <code>s</code> ; restituisce <code>s</code> .
<code>char *strncat(s,ct,n)</code>	Concatena al massimo <code>n</code> caratteri della stringa <code>ct</code> alla stringa <code>s</code> , terminando <code>s</code> con '\0'; restituisce <code>s</code> .
<code>int strcmp(cs,ct)</code>	Confronta la stringa <code>cs</code> con la stringa <code>ct</code> ; restituisce <0 se <code>cs</code> < <code>ct</code> , 0 se <code>cs</code> = <code>ct</code> , o >0 se <code>cs</code> > <code>ct</code> .
<code>int strncmp(cs,ct,n)</code>	Confronta al massimo <code>n</code> caratteri della stringa <code>cs</code> con la stringa <code>ct</code> ; restituisce <0 se <code>cs</code> < <code>ct</code> , 0 se <code>cs</code> = <code>ct</code> , o >0 se <code>cs</code> > <code>ct</code> .
<code>char *strchr(cs,c)</code>	Restituisce un puntatore alla prima occorrenza di <code>c</code> in <code>cs</code> , o NULL se <code>c</code> non compare in <code>cs</code> .
<code>char * strrchr(cs,c)</code>	Restituisce un puntatore all'ultima occorrenza di <code>c</code> in <code>cs</code> , o NULL se <code>c</code> non compare in <code>cs</code> .
<code>size_t strspn(cs,ct)</code>	Restituisce la lunghezza del prefisso di <code>cs</code> consistente di caratteri in <code>ct</code> .
<code>size_t strcspn(cs,ct)</code>	Restituisce la lunghezza del prefisso di <code>cs</code> consistente di caratteri <i>non</i> in <code>ct</code> .
<code>char *strupr(cs,ct)</code>	Restituisce un puntatore alla prima occorrenza nella stringa <code>cs</code> di qualunque carattere della stringa <code>ct</code> , o NULL se nessuno di essi compare in <code>cs</code> .
<code>char *strstr(cs,ct)</code>	Restituisce un puntatore alla prima occorrenza della stringa <code>ct</code> in <code>cs</code> , o NULL se <code>ct</code> non compare in <code>cs</code> .
<code>size_t strlen(cs)</code>	Restituisce la lunghezza di <code>cs</code> .
<code>char *strerror(n)</code>	Restituisce un puntatore alla stringa definita dall'implementazione che corrisponde all'errore <code>n</code> .
<code>char *strtok(s,ct)</code>	Ricerca in <code>s</code> i token delimitati dai caratteri di <code>ct</code> .

Una serie di chiamate a `strtok(s,ct)` scinde s in token, ognuno delimitato da un carattere di ct. Alla prima invocazione nella serie, la funzione ha argomento s diverso da `NULL`; trova il primo token di s che consista di caratteri non in ct; termina sostituendo il carattere successivo di s con '\0', restituendo un puntatore al token. Le chiamate successive, segnalate da un argomento s pari a `NULL`, restituiscono il successivo token di questo tipo, cercandolo a partire dal punto della stringa immediatamente dopo la fine del token precedente. La funzione restituisce `NULL` quando non trova più alcun token. La stringa ct può variare a seconda di ciascuna chiamata.

Le funzioni del gruppo `mem...` sono progettate per la manipolazione di oggetti alla stregua di vettori di caratteri, con l'intento di fornire un'interfaccia per funzioni efficienti. Nello schema seguente, s e t sono di tipo `void *`, cs e ct di tipo `const void *`, n di tipo `size_t`, e c è un intero convertito in `unsigned char`.

<code>void *memcpy(s,ct,n)</code>	Copia n caratteri da ct a s, e restituisce s.
<code>void *memmove(s,ct,n)</code>	Come <code>memcpy</code> , eccetto che funziona anche se gli oggetti sono parzialmente sovrapposti.
<code>int memcmp(cs,ct,n)</code>	Confronta i primi n caratteri di cs con ct; restituisce valori analoghi a quelli di <code>strcmp</code> .
<code>void *memchr(cs,c,n)</code>	Restituisce un puntatore alla prima occorrenza di c in cs, o <code>NULL</code> se c non compare fra i primi n caratteri di cs.
<code>void *memset(s,c,n)</code>	Colloca c nei primi n caratteri di s; restituisce s.

## B.4 Funzioni matematiche: <math.h>

L'intestazione `<math.h>` definisce funzioni e macro di natura matematica.

Le macro `EDOM` e `ERANGE` (che si trovano in `<errno.h>`) sono costanti intere non nulle usate per segnalare errori relativi al dominio e all'immagine delle funzioni; `HUGE_VAL` ("valore enorme") è un valore `double` positivo. Un *errore relativo al dominio* si verifica quando un argomento giace al di fuori del dominio di definizione della funzione; in questa eventualità, `errno` è impostato a `EDOM`, mentre il valore restituito dipende dall'implementazione. Un *errore relativo all'immagine* si verifica quando il risultato della funzione non può essere rappresentato come `double`. Se il risultato è troppo grande (overflow), la funzione restituisce `HUGE_VAL` con il segno appropriato, e impone `errno` a `ERANGE`. Se invece il risultato è troppo piccolo, cioè troppo vicino allo zero per essere rappresentato (underflow), la funzione restituisce zero; in questo caso, sarà l'implementazione a determinare l'eventuale impostazione di `errno` a `ERANGE`.

Nell'elenco seguente, x e y sono di tipo `double`, n è `int` e tutte le funzioni restituiscono `double`. Gli angoli nelle funzioni trigonometriche sono espressi in radianti.

<code>sin(x)</code>	Seno di x
<code>cos(x)</code>	Coseno di x
<code>tan(x)</code>	Tangente di x
<code>asin(x)</code>	Arcoseno di x; valori in $[-\pi/2, \pi/2]$ , $x \in [-1,1]$
<code>acos(x)</code>	Arcocoseno di x; valori in $[0, \pi]$ , $x \in [-1,1]$

<code>atan(x)</code>	Arcotangente di x; valori in $[-\pi/2, \pi/2]$
<code>atan2(y,x)</code>	Arcotangente di $y/x$ ; valori in $[-\pi, \pi]$
<code>sinh(x)</code>	Seno iperbolico di x
<code>cosh(x)</code>	Coseno iperbolico di x
<code>tanh(x)</code>	Tangente iperbolica di x
<code>exp(x)</code>	Funzione esponenziale $e^x$
<code>log(x)</code>	Logaritmo naturale $\ln(x)$ , $x > 0$
<code>log10(x)</code>	Logaritmo in base 10 $\log_{10}(x)$ , $x > 0$
<code>pow(x,y)</code>	$x^y$ . Un errore relativo al dominio si verifica se $x=0$ e $y \leq 0$ , o se $x < 0$ e y non è intero. $\sqrt{x}$ , $x \geq 0$
<code>sqrt(x)</code>	Minimo intero non minore di x, come <code>double</code>
<code>ceil(x)</code>	Massimo intero non maggiore di x, come <code>double</code>
<code>floor(x)</code>	Valore assoluto $ x $
<code>fabs(x)</code>	$x \cdot 2^n$
<code>ldexp(x,n)</code>	Parte intera e frazionaria esponenziale di x. La parte frazionaria è una frazione normalizzata nell'intervallo $[1/2,1]$ , ed è restituita; la parte intera, una potenza di 2, è memorizzata in *exp. Se x è zero, entrambe le parti sono zero.
<code>modf(x, double *ip)</code>	Parte intera e frazionaria di x, entrambe con lo stesso segno di x. Memorizza la parte intera in *ip, e restituisce la parte frazionaria. Resto con virgola mobile di x/y, con lo stesso segno di x. Se y è zero, il risultato dipende dall'implementazione.
<code>fmod(x,y)</code>	

## B.5 Funzioni di utilità: <stdlib.h>

L'intestazione `<stdlib.h>` definisce funzioni per le conversioni fra numeri, l'allocazione di memoria e altre operazioni simili.

<code>double atof(const char *s)</code>	Converte s in <code>double</code> ; equivale a <code>strtod(s, (char**)NULL)</code> .
<code>int atoi(const char *s)</code>	Converte s in <code>int</code> ; equivale a <code>(int)strtol(s, (char**)NULL, 10)</code> .
<code>long atol(const char *s)</code>	Converte s in <code>long</code> ; equivale a <code>strtol(s, (char**)NULL, 10)</code> .
<code>double strtod(const char *s, char **endp)</code>	Converte il prefisso di s in <code>double</code> , ignorando gli spazi iniziali; memorizza in *endp un puntatore a un eventuale suffisso non convertito, a meno che endp non sia <code>NULL</code> . Se il valore assoluto del risultato è troppo grande (overflow), restituisce <code>HUGE_VAL</code> , con il segno appropriato; se il risultato è troppo piccolo (underflow), restituisce zero. In entrambi i casi, <code>errno</code> è impostato a <code>ERANGE</code> .

## B.6 Diagnostica: <assert.h>

La macro `assert` ha fini diagnostici:

```
void assert(int espressione)
```

Se *espressione* è zero al momento dell'esecuzione di

```
assert(espressione)
```

la macro `assert` visualizzerà su `stderr` un messaggio come

```
Assertion failed: espressione, file nomefile, line nnn
```

La macro chiama poi `abort` per terminare l'esecuzione. Il nome del file sorgente e il numero della riga sono forniti dalle macro `__FILE__` e `__LINE__`.

Se `NDEBUG` risulta definita al momento dell'inclusione di `<assert.h>`, la macro `assert` è ignorata.

## B.7 Liste di argomenti di lunghezza variabile: <stdarg.h>

L'intestazione `<stdarg.h>` fornisce gli strumenti per scandire una lista di argomenti di tipo e lunghezza ignoti.

Si supponga che *ultarg* sia l'ultimo parametro non anonimo della funzione *f*, che possiede un numero variabile di argomenti. Si dichiari, all'interno di *f*, una variabile *ap* del tipo `va_list` che punterà a turno ad ogni argomento:

```
va_list ap;
```

La variabile *ap* deve essere inizializzata una volta tramite la macro `va_start` prima di accedere agli argomenti anonimi:

```
va_start(ap, ultarg);
```

Da qui in poi, ogni applicazione della macro `va_arg` produrrà un risultato il cui tipo e valore coincidano con il successivo argomento anonimo della funzione, e modificherà anche *ap* in modo che la chiamata successiva di `va_arg` restituisca l'argomento successivo:

```
tipo va_arg(va_list ap, tipo);
```

La macro

```
void va_end(va_list ap);
```

deve essere chiamata una volta dopo l'elaborazione degli argomenti, ma prima della terminazione di *f*.

## B.8 Salti non locali: <setjmp.h>

Le dichiarazioni nell'intestazione `<setjmp.h>` offrono la possibilità di evitare l'ordinaria successione chiamata di funzione-restituzione di un valore. Solitamente, ciò serve per restituire

immediatamente il controllo da una funzione annidata a livelli molto profondi del codice a un livello più esterno.

```
int setjmp(jmp_buf env)
```

Questa macro salva in *env* le informazioni relative allo stato corrente, a beneficio di `longjmp`. Il valore restituito è zero per una chiamata diretta a `setjmp`, e diverso da zero per una chiamata mediata da `longjmp`. Un'invocazione di `setjmp` è ammessa solo in certi contesti, sostanzialmente nelle condizioni di `if`, `switch` e cicli, e solo come parte di semplici espressioni relazionali.

```
if (setjmp(env) == 0)
    /* arriva qui se la chiamata e' diretta */
else
    /* arriva qui se la chiamata e' a longjmp */
```

```
void longjmp(jmp_buf env, int val)
```

Ripristina lo stato salvato tramite la chiamata più recente a `setjmp`, sfruttando le informazioni custodite da *env*; l'esecuzione prosegue come se `setjmp` fosse stata appena eseguita, con valore restituito *val* non nullo. La funzione all'interno della quale risiede `setjmp` non deve avere terminato. Gli oggetti accessibili hanno il medesimo valore che avevano al momento della chiamata a `longjmp`, eccetto che le variabili automatiche non volatili della funzione che invoca `setjmp` diverranno indefinite se sono state modificate dopo la chiamata a `setjmp`.

## B.9 Segnali: <signal.h>

L'intestazione `<signal.h>` mette a disposizione degli strumenti per trattare condizioni eccezionali che abbiano luogo durante l'esecuzione, come per esempio la ricezione di un segnale di interruzione (un interrupt) proveniente da una fonte esterna, o un errore durante l'esecuzione stessa.

```
void (*signal(int sig, void (*handler)(int)))(int)
```

Stabilisce come gestire gli eventuali segnali ricevuti in seguito. Se *handler* (il gestore) è `SIG_DFL`, si adotta il comportamento di default definito dall'implementazione; se è `SIG_IGN`, si ignora il segnale; altrimenti, si chiama la funzione puntata da *handler*, con argomento il segnale. I segnali validi includono:

<code>SIGABRT</code>	terminazione anomala, per esempio causata da <code>abort</code>
<code>SIGFPE</code>	errore aritmetico, per esempio divisione per zero o overflow
<code>SIGILL</code>	immagine della funzione illegale, per esempio istruzione illegale
<code>SIGINT</code>	attenzione all'interazione richiesta, per esempio interruzione
<code>SIGSEGV</code>	accesso illegale alla memoria, per esempio tentativo di accesso al di fuori dei limiti della memoria
<code>SIGTERM</code>	richiesta di terminazione inviata a questo programma

Restituisce il precedente valore di *handler* relativo al segnale in questione, o `SIG_ERR` se si verifica un errore.

Quando, successivamente all'invocazione di `signal`, si riceve un segnale `sig`, si ripristina il comportamento di default del segnale; poi si invoca il gestore del segnale, come se si chiamasse `(*handler)(sig)`. Se il gestore termina, l'esecuzione riprende dal punto in cui era stato rilevato il segnale.

Lo stato iniziale dei segnali è dipendente dall'implementazione.

`int raise(int sig)`

Trasmette il segnale `sig` al programma; restituisce un valore non nullo in caso di insuccesso.

## B.10 Funzioni di data e ora: <time.h>

L'intestazione `<time.h>` definisce funzioni e tipi dedicati alla manipolazione di date e orari. Alcune funzioni trattano l'*ora locale*, che può essere diversa dal cosiddetto *tempo del calendario* (*calendar time*), per esempio a causa del diverso fuso orario. I tipi aritmetici `clock_t` e `time_t` rappresentano l'ora; la struttura `struct tm` raccoglie le informazioni relative al tempo del calendario:

<code>int tm_sec;</code>	secondi dopo il minuto (0, 61)
<code>int tm_min;</code>	minuti dopo l'ora (0, 59)
<code>int tm_hour;</code>	ore dalla mezzanotte (0, 23)
<code>int tm_mday;</code>	giorno del mese (1, 31)
<code>int tm_mon;</code>	mesi a partire da Gennaio (0, 11)
<code>int tm_year;</code>	anni a partire dal 1900
<code>int tm_wday;</code>	giorni della settimana a partire da domenica (0, 6)
<code>int tm_yday;</code>	giorni dell'anno a partire dall'1 Gennaio (0, 365)
<code>int tm_isdst;</code>	opzione per l'ora legale

L'opzione `tm_isdst` contiene un valore positivo se è in vigore l'ora legale, zero se non lo è, e negativo se l'informazione non è disponibile.

`clock_t clock(void)`

Restituisce il tempo di CPU usato dal programma dall'inizio dell'esecuzione, o -1 se l'informazione non è disponibile; `clock() /CLOCKS_PER_SEC` è in secondi.

`time_t time(time_t *tp)`

Restituisce l'ora corrente del calendario, o -1 se l'informazione non è disponibile. Se `tp` non è `NULL`, il valore restituito è anche assegnato a `*tp`.

`double difftime(time_t time2, time_t time1)`

Restituisce `time2-time1` espresso in secondi.

`time_t mktime(struct tm *tp)`

Converte l'ora locale contenuta nella struttura `*tp` nell'ora corrispondente al tempo del calendario nello stesso formato usato da `time`. Le componenti avranno valori corrispondenti agli intervalli di tempo mostrati nello schema precedente. Restituisce il tempo del calendario corrispondente, o -1 se i valori non sono rappresentabili.

Le prossime quattro funzioni restituiscono puntatori a oggetti statici i cui contenuti possono essere riscritti da altre chiamate.

`char *asctime(const struct tm *tp)`

Converte l'ora in una struttura `*tp` in una stringa della forma  
`Sun Jan 3 15:14:13 1988\n\0`

`char *ctime(const time_t *tp)`

Converte il tempo del calendario `*tp` in ora locale; equivale a `asctime(localtime(tp))`

`struct tm *gmtime(const time_t *tp)`

Converte il tempo del calendario `*tp` nel formato noto come Coordinate Universali di Tempo (UTC, *Universal Coordinated Time*). Restituisce `NULL` se UTC non è disponibile. Il nome `gmtime` deriva da Greenwich Mean Time (Ora di Greenwich) come una volta si definiva l'UTC.

`struct tm *localtime(const time_t *tp)`

Converte il tempo del calendario `*tp` in ora locale.

`size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)`

Imposta nella stringa `s` la rappresentazione dei valori relativi a data e ora contenuti in `*tp`, secondo le specifiche fornite in `fmt`, che è analogo al parametro `format` di `printf`. I caratteri ordinari (incluso lo '\0' finale) si copiano in `s`. Ogni occorrenza di `%c`, dove `c` è un carattere, è sostituita nel modo descritto nello schema mostrato di seguito, con valori locali appropriati. Non più di `smax` caratteri sono copiati in `s`. Restituisce il numero di caratteri, escluso '\0', o zero se sono stati generati più di `smax` caratteri.

`%a` nome abbreviato del giorno della settimana

`%A` nome completo del giorno della settimana

`%b` nome abbreviato del mese

`%B` nome del mese per esteso

`%c` rappresentazione dell'ora e della data locali

`%d` giorno del mese (01-31)

`%H` ora (00-23)

`%I` ora (01-12)

`%j` giorno dell'anno (001-366)

`%m` mese (01-12)

`%M` minuto (00-59)

`%p` equivalente locale di AM o PM (antimeridiano o pomeridiano)

`%S` secondo (00-61)

`%U` numero della settimana dell'anno, con domenica come primo giorno della settimana (00-53)

`%w` giorno della settimana (0-6, domenica è 0)

`%W` numero della settimana dell'anno, con lunedì come primo giorno della settimana (00-53)

`%x` rappresentazione della data locale

`%X` rappresentazione dell'ora locale

%y	anno del secolo (00-99)
%Y	anno
%Z	fuso orario
%%	%

## B.11 Limiti definiti dall'implementazione: <limits.h> e <float.h>

L'intestazione `<limits.h>` definisce alcune costanti relative alle dimensioni dei tipi interi. I valori riportati nello schema seguente sono le magnitudini minime ammesse; sono possibili valori maggiori.

CHAR_BIT	8	numero di bit di char
CHAR_MAX	UCHAR_MAX	o
	SCHAR_MAX	valore massimo di char
CHAR_MIN	0	o SCHAR_MIN
		valore minimo di char
INT_MAX	+32767	valore massimo di int
INT_MIN	-32767	valore minimo di int
LONG_MAX	+2147483647	valore massimo di long
LONG_MIN	-2147483647	valore minimo di long
SCHAR_MAX	+127	valore massimo di signed char
SCHAR_MIN	-127	valore minimo di signed char
SHRT_MAX	+32767	valore massimo di short
SHRT_MIN	-32767	valore minimo di short
UCHAR_MAX	255	valore massimo di unsigned char
UINT_MAX	65535	valore massimo di unsigned int
ULONG_MAX	4294967295	valore massimo di unsigned long
USHRT_MAX	65535	valore massimo di unsigned short

I nomi che compaiono nello schema seguente, un sottoinsieme di `<float.h>`, sono costanti relative all'aritmetica con virgola mobile. I valori, laddove presenti, devono essere intesi come la magnitudine minima ammessa della quantità corrispondente. Le varie implementazioni definiscono valori appropriati.

FLT_RADIX	2	base della rappresentazione esponenziale, per esempio 2, 16
FLT_ROUNDS		modalità dell'arrotondamento per l'addizione con virgola mobile
FLT_DIG	6	precisione singola: numero di cifre decimali
FLT_EPSILON	1E-5	minimo numero $x$ tale che $1.0 + x \neq 1.0$
FLT_MANT_DIG		numero di cifre della mantissa, in base FLT_RADIX
FLT_MAX	1E+37	massimo numero con virgola mobile
FLT_MAX_EXP		massimo $n$ tale che $\text{FLT\_RADIX}^n - 1$ sia rappresentabile
FLT_MIN	1E-37	minimo numero con virgola mobile normalizzato
FLT_MIN_EXP		minimo $n$ tale che $10^n$ sia un numero normalizzato
DBL_DIG	10	precisione multipla: numero di cifre decimali
DBL_EPSILON	1E-9	precisione multipla: minimo numero $x$ tale che $1.0 + x \neq 1.0$

DBL_MANT_DIG		precisione multipla: numero di cifre della mantissa, in base FLT_RADIX
DBL_MAX	1E+37	massimo numero double con virgola mobile
DBL_MAX_EXP		massimo $n$ tale che $\text{FLT\_RADIX}^n - 1$ sia rappresentabile come double
DBL_MIN	1E-37	minimo numero double con virgola mobile normalizzato
DBL_MIN_EXP		minimo $n$ tale che $10^n$ sia un numero double normalizzato

# C

## Sommario delle modifiche

LA DEFINIZIONE DEL LINGUAGGIO C ha subito alcuni cambiamenti successivi alla pubblicazione della prima edizione di questo libro. Quasi tutte le modifiche riguardavano estensioni del linguaggio originale, accuratamente studiate al fine di garantire compatibilità con la prassi di programmazione già affermata; alcune ponevano rimedio a qualche ambiguità nella descrizione originale; altre, infine, implicavano dei mutamenti alla prassi esistente. Molte nuove funzionalità vennero annunciate nella documentazione allegata ai compilatori disponibili dalla AT&T, e furono poi adottate anche da altri produttori di compilatori C. Più di recente, la commissione ANSI preposta all'emanazione di uno standard per il linguaggio C ha annesso molte di queste modifiche, introducendone anche altre di rilievo. Alcuni compilatori commerciali hanno tenuto conto del lavoro della commissione persino prima del rilascio ufficiale dello standard.

Questa appendice riassume le discrepanze fra il linguaggio definito dalla prima edizione del libro e ciò che si ritiene sarà lo standard ANSI del C. Tratteremo solo il linguaggio in sé, e non l'ambiente o la libreria: aspetti dello standard senz'altro importanti, ma irrilevanti ai nostri fini, in quanto la prima edizione del libro non si occupava né dell'ambiente esecutivo, né della libreria.

- Rispetto alla prima edizione, il preprocessore è stato definito con più cura dallo standard, e le sue funzionalità risultano estese: esso si basa esplicitamente sui token; prevede nuovi operatori per la concatenazione dei token (##) e la creazione di stringhe (#); prevede nuove direttive come #elif e #pragma; consente esplicitamente la ridichiarazione di macro con la stessa sequenza di token; non sostituisce più i parametri all'interno delle stringhe. È ora possibile spezzare le righe tramite \, e non solo nelle stringhe o nelle definizioni di macro. Si veda §A.12.
- Il numero minimo di lettere significative per tutti gli identificatori interni è stato portato a 31; per gli identificatori esterni, la soglia minima rimane di 6 lettere, senza distinzione fra maiuscole e minuscole. (Molte implementazioni alzano la soglia minima.)
- Le sequenze triplici, che iniziano con ??, permettono di rappresentare caratteri assenti in alcuni set di caratteri. Sono state definite sequenze di controllo per rappresentare #\^\{\}\~ (si veda §A.12.1). Si noti che l'introduzione delle sequenze triplici può modificare il senso delle stringhe contenenti ??.
- Sono state definite nuove parole chiave (`void`, `const`, `volatile`, `signed`, `enum`). La vecchia parola chiave `entry`, fallita in partenza, è stata eliminata.
- Sono state definite nuove sequenze di controllo da usare all'interno delle costanti stringa o delle costanti carattere. L'effetto di un'occorrenza di \ seguita da un carattere che non dia luogo a una sequenza di controllo è indefinito. Si veda §A.2.5.2.
- È ora possibile riscontrare la modifica più semplice ambita da tutti: 8 e 9 non sono cifre in notazione ottale.
- Lo standard definisce un insieme allargato di suffissi che specifichino esplicitamente il tipo di una costante: U o L per gli interi, F o L per i numeri con virgola mobile. Lo standard raffina anche le regole per la determinazione del tipo delle costanti prive di suffisso (§A.2.5).
- Le costanti stringa adiacenti vengono concatenate.
- Esiste ora una notazione per definire stringhe di caratteri estesi e costanti carattere estese; si veda §A.2.6.
- I caratteri, e anche gli altri tipi, possono essere esplicitamente dichiarati con o senza segno, tramite le parole chiave `signed` e `unsigned`. La locuzione `long float` è stata eliminata come parafrasi di `double`, ma `long double` è ammessa per dichiarare una quantità numerica con virgola mobile con precisione estesa.
- Il tipo `unsigned char` è disponibile ormai da diverso tempo. Lo standard introduce la parola chiave `signed` per rendere esplicita la presenza del segno, sia per il tipo `char` che per altri oggetti interi.
- Il tipo `void` è disponibile già da alcuni anni nella maggioranza delle implementazioni. Lo standard definisce adesso il tipo `void *` come tipo di puntatore generico, in luogo del precedente `char *`. Al contempo, lo standard emana l'esplicita

proibizione di usare promiscuamente puntatori e interi, e puntatori di tipi diversi, a meno di non ricorrere alla conversione forzata (cast).

- Lo standard stabilisce esplicitamente il limite inferiore dei valori dei tipi aritmetici, e prescrive delle intestazioni (`<limits.h>` e `<float.h>`) che rendano conto delle caratteristiche della specifica implementazione.
- Le enumerazioni sono una novità rispetto alla prima edizione di questo libro.
- Lo standard adotta la nozione di attributi dei tipi del C++, per esempio `const` (§A.8.2).
- Le stringhe non sono più modificabili; si possono quindi collocare nella memoria a sola lettura.
- Le "conversioni aritmetiche abituali" sono cambiate. In sostanza, si è passati dalla considerazione "unsigned vince sempre per gli interi; per la virgola mobile, si usa sempre double" al nuovo motto "si promuova al più piccolo tipo di dimensione sufficiente". Si veda §A.6.5.
- I vecchi operatori di assegnamento come =+ sono davvero scomparsi. Inoltre, gli operatori di assegnamento sono adesso dei token singoli: nella prima edizione, erano coppie di token, e potevano essere separati da spazi.
- È revocata la licenza dei compilatori di trattare operatori associativi in senso matematico come operatori associativi in senso computazionale.
- Per simmetria con l'operatore unario +, è stato ora introdotto un operatore unario -.
- Un puntatore a una funzione può adesso fungere da designatore di una funzione senza l'esplicita presenza dell'operatore \*. Si veda §A.7.3.2.
- Le strutture possono essere assegnate, passate alle funzioni e restituite dalle funzioni.
- È concesso applicare l'operatore & (indirizzo dell'operando) ai vettori; il risultato è un puntatore al vettore.
- L'operatore `sizeof`, nella prima edizione, restituiva il tipo `int`; in seguito, molte implementazioni hanno adottato `unsigned int`. Lo standard rende il tipo di `sizeof`, detto `size_t`, esplicitamente dipendente dall'implementazione, ma richiede che esso sia definito nell'intestazione standard `<stddef.h>`. Una modifica simile si applica al tipo della differenza fra due puntatori (`ptrdiff_t`). Si vedano §A.7.4.8 e §A.7.7.
- L'operatore & non può essere applicato a un oggetto di classe `register`, nemmeno qualora l'implementazione decida di non mantenere effettivamente l'oggetto in un registro.
- Il tipo di un'espressione che coinvolga l'operatore di scorrimento è quello dell'operando sinistro; l'operando destro non può influire sul tipo del risultato. Si veda §A.7.8.

- Lo standard rende legale la creazione di un puntatore che punta appena dopo la fine di un vettore, e consente di eseguire su di esso confronti e operazioni aritmetiche. Si veda §A.7.7.
- Ispirandosi al C++, lo standard introduce la nozione di dichiarazione del prototipo di una funzione, che incorpora i tipi dei parametri; definisce inoltre esplicitamente le funzioni con liste di argomenti di lunghezza variabile, e specifica le modalità di trattamento degli argomenti. Si vedano §§A.7.3.2, A.8.6.3, B.7. Il vecchio stile è ancora valido, con certe restrizioni.
- Lo standard proibisce le dichiarazioni vuote, cioè prive di un dichiaratore e che non dichiarino almeno una struttura, un'unione o un'enumerazione. D'altro canto, la dichiarazione del solo contrassegno di una struttura o unione dichiara nuovamente il contrassegno, anche qualora esso sia già visibile in quel punto del codice.
- Sono proibite le dichiarazioni esterne dei dati prive di specificatori o qualificatori (dichiaratori nudi e crudi, insomma).
- Alcune implementazioni esportavano all'intero file sorgente le dichiarazioni `extern` che comparivano in blocchi interni. Lo standard chiarisce che il campo di visibilità di tali dichiarazioni è il blocco in cui compaiono.
- Le variabili dichiarate al livello più esterno del codice che implementa una funzione non possono fare ombra ai suoi parametri, perché il campo di visibilità di questi ultimi è definito essere il corpo della funzione.
- Lo spazio dei nomi degli identificatori è stato in parte modificato. Lo standard raccoglie tutti i contrassegni in un singolo spazio dei nomi, e prevede un distinto spazio dei nomi per le etichette; si veda §A.11.1. Inoltre, i nomi dei membri sono ora associati alla struttura o unione di cui fanno parte. (Questa era la prassi comune già da tempo.)
- Le unioni possono essere inizializzate; l'inizializzatore si riferisce al primo membro.
- Le strutture e le unioni automatiche, così come i vettori automatici, possono essere inizializzate, sebbene con alcune restrizioni.
- I vettori di caratteri con una dimensione esplicita possono essere inizializzati tramite una costante stringa di quella esatta dimensione (in questa circostanza si ignora '\0').
- Le etichette dei casi e la condizione del costrutto `switch` possono avere un qualunque tipo intero.

## Indice analitico

- , operatore di negazione, 40, 213  
`!=`, operatore di disegualanza, 13, 39, 215  
`##`, operatore preprocessore, 88, 244  
`#`, operatore preprocessore, 88, 244  
`#define`, 11, 87, 243  
 con argomenti, 87  
 su più righe, 87  
`vs. enum`, 38,  
`#else, #elif`, 89, 246  
`#endif`, 89  
`#error`, 247  
`#if`, 89, 136, 246  
`#ifdef`, 89, 247  
`#ifndef`, 89, 247  
`#include`, 30, 86, 155, 245  
`#line`, 247  
`#pragma`, 247  
`#undef`, 88, 176, 243  
`%`, operatore modulo, 39, 213  
`%ld`, specifica di conversione, 15  
`&&`, operatore AND logico, 18, 40, 47, 216  
`&`, operatore AND bit per bit, 46, 216  
`&`, operatore di indirizzo, 92, 212  
`*`, operatore di indirezione, 92, 212  
`*`, operatore di moltiplicazione, 39, 213  
`'`, carattere apice, 16, 35, 200  
`"`, carattere virgolette, 3, 4, 11, 16, 31, 36, 200  
`-`, operatore di decremento, 14, 44, 45, 105, 211  
`-`, operatore di sottrazione, 39, 214  
`-`, operatore meno unario, 212  
`,`, operatore virgola, 60, 218  
`.`, operatore membro di struttura, 129, 208  
`...`, dichiarazione, 158, 210  
`.h`, suffisso del nome di un file, 30  
`/`, operatore di divisione, 7, 39, 214  
`?:`, espressione condizionale, 49, 217  
`\``, carattere barra inversa, 4, 36  
`\0`, carattere nullo, 27, 38, 200  
`\a`, allarme, 36, 200  
`\b`, carattere backspace, 4, 36, 200  
`\f`, carattere salto pagina, 36, 200  
`\n`, carattere newline, 3, 12, 16, 36, 200, 258  
`\ooo...`, costante in base otto (ottale), 36, 200  
`\r`, ritorno del carrello, 36, 200  
`\t`, carattere di tabulazione, 4, 7, 36, 200

\v, carattere di tabulazione verticale, 36, 200  
 \xhh, sequenza di controllo esadecimale, 35, 200  
 \xhh, costante in base sedici (esadecimale), 36, 200  
 ^, operatore OR esclusivo bit per bit, 46, 216  
 \_, carattere di sottolineatura (underscore), 34, 198, 258  
 \_FILE\_, nome del preprocessore, 272  
 \_LINE\_, nome del preprocessore, 272  
 fillbuf, funzione, 182  
 IOPBF, IOLBF, IONBF, 260  
 !, operatore OR inclusivo bit per bit, 46, 216  
 !!, operatore OR logico, 18, 40, 47, 217  
 -, tilde, operatore di complemento a uno, 47, 212  
 +, operatore di addizione, 39, 214  
 +, operatore più unario, 212  
 ++, operatore di incremento, 14, 44, 45, 105, 211  
 <, operatore minore di, 39, 215  
 <<, operatore di scorrimento a sinistra, 47, 214-215  
 <=, operatore minore o uguale di, 39, 215  
 <assert.h>, intestazione, 272  
 <cctype.h>, intestazione, 41, 266  
 <errno.h>, intestazione, 266  
 <float.h>, intestazione, 35, 276  
 <limits.h>, intestazione, 35, 276  
 <locale.h>, intestazione, 257  
 <math.h>, intestazione, 42, 268  
 <setjmp.h>, intestazione, 272-273  
 <signal.h>, intestazione, 273-274  
 <stdarg.h>, intestazione, 158, 172, 272  
 <stddef.h>, intestazione, 100, 101, 135, 257  
 <stdio.h>  
     contenuti di 3, 180  
     intestazione, 13, 86, 153, 154, 155, 257  
 <stdlib.h>, intestazione, 69, 143, 269  
 <string.h>, intestazione, 37, 104, 105, 267  
 <time.h>, intestazione, 274  
 =, operatore di assegnamento, 14, 40, 217-218  
 ==, operatore di uguaglianza, 16, 39, 215-216  
 >, operatore maggiore di, 39, 215

->, operatore puntatore membro di struttura, 132, 209  
 >=, operatore maggiore o uguale di, 39, 215  
 >>, operatore di scorrimento a destra, 47, 214-215  
**A**  
 a.out, 2, 69  
 abort, funzione della libreria, 270  
 abs, funzione della libreria, 271  
 acos, funzione della libreria, 268  
 addizione, operatore di +, 39, 214  
 addpoint, funzione, 130  
 addtree, funzione, 141  
 albero binario, 139  
 albero sintattico, 122  
 allarme, \a, 36, 200  
 allineamento tramite union, 191  
 allineamento, di un campo di bit, 150, 223  
 alloc, funzione, 99  
 allocatore, di memoria, 142, 190-195  
 allocazione di memoria, 220  
 annidate  
 apice, ', 16, 35, 200  
 argc, contatore degli argomenti, 112  
 argomento attuale *si veda* argomento  
 argomento  
     dalla riga di comando, 112-117  
     definizione, 22, 209  
     di funzione, 22, 210  
     lista di, di lunghezza variabile, 157, 178, 210, 229, 238, 272  
     lista di argomenti void, 30, 71, 229, 238  
     promozione di, 43, 210  
     puntatore, 95  
     sottovettore, 98  
 argv, vettore degli argomenti, 116, 154, 164  
 aritmetica degli indirizzi *si veda* aritmetica  
     di puntatori  
 aritmetica di puntatori, 97, 99-102, 138, 214  
 aritmetici, tipi, 203  
 ASCII, set di caratteri, 16, 35, 41, 242, 266  
 asctime, funzione della libreria, 275  
 asin, funzione della libreria, 268  
 asm, parola chiave, 198  
 assegnamento  
     conversione tramite, 42, 217-218  
     espressione, 13-14, 18, 49, 217-218

istruzioni annidate di, 14, 17, 51  
 multiplo, 17  
 operatore di, +=, 48  
 operatore di, =, 14, 40, 217  
 operatori di, 40, 48, 217  
 soppressione dello, scanf, 160, 262  
 associatività degli operatori, 50, 207  
 atan, atan2, funzioni della libreria, 269  
 atexit, funzione della libreria, 271  
 atof, funzione, 69  
 atof, funzione della libreria, 269  
 atoi, funzione, 41, 59, 71  
 atoi, funzione della libreria, 269  
 atol, funzione della libreria, 269  
 auto, specificatore della classe di  
     memorizzazione, 220  
 automatica, classe di memorizzazione, 28, 202  
 automatica, variabile, 28, 72, 78, 81, 202  
 automatiche, inizializzazione di variabili, 28, 38, 83, 230  
 automatiche, visibilità di variabili, 78, 241  
 autoreferenziale, struttura, 139, 223  
**B**  
 barra inversa, carattere di backslash, \\ 4, 36  
 bidimensionali, vettori, 109, 110, 231  
     inizializzazione di vettori, 110, 231  
 binario, albero, 139  
 binario, flusso, 163, 258  
 binsearch, funzione, 56, 134, 137  
 bisestile, computazione dell'anno, 39, 109  
 bit per bit  
     operatore AND, &, 46, 216  
     operatore OR esclusivo, ^, 46, 216  
     operatore OR inclusivo, !, 46, 216  
     operatori, 46, 216  
 bit, idiom per la manipolazione di, 46-47  
 bitcount, funzione, 48  
 blocchi, struttura a, 53, 82, 235  
 blocco *si veda* istruzione composta  
 blocco, inizializzazione in, 82, 235  
 break, istruzione, 57, 62, 237  
 bsearch, funzione della libreria, 271  
 buffer  
     getchar con, 176  
     input con uso del *si veda* setbuf, setvbuf  
 BUFSIZ, 260

**C**  
 calcolatrice, programma, 70, 72, 74, 160  
 callloc, funzione della libreria, 170, 270  
 campanello, carattere *si veda* \a  
 campo di bit  
     allineamento di, 150, 223  
     dichiarazione di, 150, 222  
 campo di visibilità, 201, 240-242  
     lessicale, 241  
 canonrect, funzione, 131  
 carattere  
     con segno, 42, 202  
     costante, 16, 35, 199  
     estesa, costante, 200  
     intero, conversioni, 9, 41, 204  
     ottale, costante, 35  
     privato di segno, 42, 202  
 caratteri  
     di spaziatura, 160, 169, 262, 266  
     EBCDIC, set di, 41  
     funzioni per l'analisi di, 169, 264  
     ingresso e uscita di, 12, 154  
     programma per il conteggio di, 14-15  
     set di, 242  
     stringa di *si veda* costante stringa  
     vettore di, 25-27, 103  
 carrello, carattere di ritorno del, 36, 200  
 case, etichetta, 56, 234  
 casi estremi nelle condizioni, 15, 63  
 cast  
     conversione tramite, 43, 206, 213  
     operatore, 43, 143, 206, 213, 231  
 cat, programma, 164, 165  
 cc, comando, 2, 69  
 ceil, funzione della libreria, 269  
 char, tipo, 6, 34, 202, 220  
 chiamata  
     per riferimento, 24  
     per valore, 24, 94, 210  
 ciclo *si veda* while, for, do  
 clearerr, funzione della libreria, 265  
 clock, funzione della libreria, 274  
 clock\_t, nome di tipo, 274  
 CLOCKS\_PER\_SEC, 274  
 close, chiamata di sistema, 178  
 closedir, funzione, 189  
 comandi, argomenti dalla riga di, 112-117  
 commenti, 5, 198, 242  
 compilare file multipli, 68

compilazione singola, 65, 78, 240  
 composta, istruzione, 53, 82, 235  
 concatenazione  
   di stringhe, 37, 88, 201  
   di token, 244  
 condizionale  
   compilazione, 89, 245  
   espressione, 49, 217  
 confronto fra puntatori, 100, 193, 215-216  
 congiunzione di righe, 243  
**const**, attributo, 39, 203, 221  
**continue**, istruzione, 63, 237  
 contrassegno  
   di enumerazione, 226  
   di struttura, 226  
   di unione, 221  
 controllo, carattere di, 266  
 conversione, 204  
   carattere-intero, 9, 41, 204  
   del nome di un vettore, 97, 207  
   di funzione, 207  
   di puntatori, 206, 214  
   **double-float**, 43, 205  
   **float-double**, 42, 205  
   intero, a virgola mobile, 9, 204-205  
   intero-carattere, 43  
   intero-puntatore, 206, 214  
   operatore esplicito di *si veda cast*  
   puntatore-intero, 206, 214  
   tramite assegnamento, 42, 217  
   tramite cast, 43, 206, 213  
   tramite **return**, 71, 237  
   virgola mobile-intero, 43, 204-205  
 conversioni aritmetiche abituali, 40, 205  
**copy**, funzione, 26, 30  
**cos**, funzione della libreria, 268  
**cosh**, funzione della libreria, 269  
 costante  
   espressione, 36, 56, 89, 218-219  
   manifesta, 244  
   suffisso di, 35, 200  
   tipo di, 35, 199  
 costanti, 35, 198  
**creat**, chiamata di sistema, 176, 177  
**ctime**, funzione della libreria, 275

**D**

**data**, conversione di, 109  
**day\_of\_year**, funzione, 109

**dcl**, funzione, 121, 122  
**dcl**, programma, 121, 122  
 decremento, operatore di, -, 14, 44, 45, 105, 211  
**default**  
   dimensione di un vettore, 84, 112, 134  
   inizializzazione, 84, 230  
**default**, etichetta, 56-57, 234  
**defined**, operatore del preprocessore, 89, 246  
**definizione**  
   di argomento, 22, 209  
   di classe di memorizzazione, 220  
   di funzione, 22, 68, 238  
   di macro, 243  
   di parametro, 22, 209-210  
   di variabile esterna, 30, 240  
   provvisoria, 240  
   rimozione di *si veda #undef*  
 deriferimento *si veda indirezione*  
 derivati, tipi, XIII, 6, 203  
 designatore di funzione, 209  
 destra, operatore di scorrimento a >>, 47, 214  
**dichiaratore**, 226  
   astratto, 232  
   di funzione, 228  
   di vettore, 227  
 dichiarazioni, 5, 38, 219-225  
   della classe di memorizzazione, 220  
   di campo di bit, 150, 222  
   di funzione, 228  
   di puntatore, 92, 98, 227  
   di struttura, 128, 221  
   di unione, 148, 221  
   di variabile esterna, 28, 238  
   di vettore, 19, 109, 227  
 esterna, 237-238  
 implicita di funzione, 28, 70, 209  
 tipo in, 226  
**typedef**, 146, 221, 233  
 vs. definizione, 30, 78-79, 219  
**difftime**, funzione della libreria, 274  
**dimensione**  
   di numeri, 6, 15, 34, 276  
   di strutture, 138, 213  
**DIR**, struttura, 185  
**dir.h**, file intestazione, 188  
**dirdcl**, funzione, 122, 123  
**directory**, programma per la lista di, 184

**Dirent**, struttura, 186, 188  
 direttiva, del preprocessore, 86, 242  
**dirwalk**, funzione, 187  
 disuguaglianza, operatore di, !=, 13, 39, 215  
**div**, funzione della libreria, 271  
**div\_t, ldiv\_t**, nomi di tipi, 271  
 divisione  
   operatore di /, 7, 39, 213  
   tra interi, 7, 39  
**do**, istruzione, 61, 237  
**double**, costante, 35, 203  
**double**, tipo, 6, 15, 34, 203, 221  
**double-float**, conversione, 43, 205

## E

**e**, notazione, 35, 201  
**EBCDIC**, set di caratteri, 41  
 eccezioni, 207  
**echo**, programma, 113  
**EDOM**, 268  
 effetti collaterali, 51, 87, 207, 210  
 efficienza, 49, 86, 142, 173  
 elenco  
   di intestazioni standard, 257  
   di parole chiave, 198  
   di sequenze di controllo, 36, 200  
**else** *si veda if-else*, istruzione  
**else-if**, 20, 55  
**enum**, specificatore, 37, 225  
   vs. **#define**, 38  
 enumeratori, 200, 225  
 enumerazione  
   contrassegno di, 226  
   costante di, 37, 89, 199, 200-201, 225  
   tipo, 203  
**EOF**, 13, 154, 259  
 equivalenza di tipo, 233  
**ERANGE**, 268  
**errno**, 266  
**error**, funzione, 178  
 errori nell'ingresso e uscita di dati, 165, 266  
 esadecimale, costante, 0x..., 35, 200  
 esadecimale, sequenza di controllo, \xhh, 36, 200  
 eseguire un programma in C, 2-3, 22  
 espansione di macro, 243  
 esplicita, conversione *si veda cast*  
 espressione, 207-209

assegnamento, 13-14, 18, 49, 217-218  
 costante, 36, 56, 89, 218-219  
 istruzione, 53, 55, 234  
 ordine di valutazione di, 50, 207  
 primaria, 208  
 tra parentesi, 208  
 esterna  
   definizione di variabile, 30, 240  
   dichiarazione, 237-238  
   dichiarazione di variabile, 28, 238  
   variabile, 28, 71, 201  
 esterne  
   inizializzazione di variabili, 38, 79, 83, 230  
   variabili **static**, 81  
   visibilità di variabili, 78, 241  
 esterni, lunghezza di nomi, 34, 198  
 esterno, linkage, 72, 198, 202, 220, 240  
 estesa  
   costante carattere, 200  
   costante stringa, 200  
 etichetta, 63, 234  
   **case**, 56, 57, 234  
   **default**, 56-57, 234  
   istruzione, 63, 234  
   visibilità di, 64, 234, 241  
 evitare **goto**, 64  
**exit**, funzione della libreria, 165, 270  
**EXIT\_FAILURE, EXIT\_SUCCESS**, 271  
**exp**, funzione della libreria, 269  
**extern**, specificatore di classe di  
   memorizzazione, 28, 30, 78, 220

## F

**fabs**, funzione della libreria, 269  
**fclose**, funzione della libreria, 165, 259  
**fcntl.h**, file intestazione, 176  
**feof**, funzione della libreria, 166, 266  
**feof**, macro, 181  
**ferror**, funzione della libreria, 166, 266  
**ferror**, macro, 181  
**fflush**, funzione della libreria, 259  
**fgetc**, funzione della libreria, 264  
**fgetpos**, funzione della libreria, 265  
**fgets**, funzione della libreria, 167, 264  
 file  
   aggiunta in coda a, 163, 179, 258  
   apertura di, 162, 174, 176  
   creazione di, 163, 177  
   inclusione di, 86, 245

modalità di accesso a, 162, 182, 258  
 permessi relativi a, 175, 177  
 programma per la concatenazione di, 154, 162, 164  
 programma per la replica di, 13, 175  
 puntatore a, 163, 179, 258  
**FILE**, nome di tipo, 163  
**filecopy**, funzione, 164  
**FILENAME\_MAX**, 259  
 fine del file *si veda EOF*  
**float**, costante, 35, 198, 200  
**float**, tipo, 6, 34, 203, 221  
**float-double**, conversione, 42, 205  
**floor**, funzione della libreria, 269  
 flusso binario, 163, 258  
 flusso di testo, 12, 153, 258  
**fmod**, funzione della libreria, 269  
 fondamentali, tipi di dati, 6, 34, 202  
**fopen**, funzione della libreria, 162, 258  
**fopen**, funzione, 181  
**FOPEN\_MAX**, 259  
**for** vs **while**, 11, 58  
**for(;;)**, ciclo infinito, 58-59, 87  
**for**, istruzione, 10, 11, 15, 58, 236  
 formale, parametro *si veda* parametro  
 formattazione  
   dell'input *si veda* **scanf**  
   dell'output *si veda* **printf**  
**fortran**, parola chiave, 198  
 forzata, conversione *si veda* cast  
**fpos\_t**, nome di tipo, 265  
**fprintf**, funzione della libreria, 164, 260  
**fputc**, funzione della libreria, 264  
**fputs**, funzione, 167  
**fputs**, funzione della libreria, 167, 264  
**fread**, funzione della libreria, 265  
**free**, funzione, 193  
**free**, funzione della libreria, 170, 270  
**freopen**, funzione della libreria, 165, 259  
**frexp**, funzione della libreria, 269  
**fscanf**, funzione della libreria, 164, 262  
**fseek**, funzione della libreria, 265  
**fsetpos**, funzione della libreria, 265  
**fsize**, funzione, 187  
**fsize**, programma, 186  
**fstat**, chiamata di sistema, 188  
**ftell**, funzione della libreria, 265  
 funzione  
   argomento di, 22, 210

conversione di argomenti di *si veda*  
   argomenti, promozione di  
 conversione di, 207  
 definizione di, 22, 68, 238  
 designatore, 209  
   di verifica di caratteri, 169, 266  
 dichiaratore di, 228  
 dichiarazione di, 228  
 dichiarazione implicita di, 23, 70, 209  
 dichiarazione static di, 81  
 lunghezza del nome di, 34, 198  
 nuovo stile, 210  
 prototipo di, 23, 43, 44, 70, 86, 118, 210  
 puntatore a, 117, 147, 209  
 semantica di chiamata a, 209  
 sintassi di chiamata a, 209  
 vecchio stile, 23, 30, 71, 210  
**fwrite**, funzione della libreria, 265

**G**

generico, puntatore *si veda void \**, puntatore  
**getbits**, funzione, 47  
**getc**, funzione della libreria, 163, 264  
**getc**, macro, 179  
**getch**, funzione, 76  
**getchar** con buffer, 176  
**getchar** senza buffer, 175  
**getchar**, funzione della libreria, 12, 154, 155, 163, 264  
**getenv**, funzione della libreria, 271  
**getint**, funzione, 94, 95  
**getline**, funzione, 26, 29, 66, 67, 69, 167  
**getop**, funzione, 75, 80  
**gets**, funzione della libreria, 167, 264  
**gettoken**, funzione, 123, 124  
**getword**, funzione, 136  
**gmtime**, funzione della libreria, 275  
**goto**, istruzione, 63, 237

**H**

**hash**, funzione, 144  
**hash**, tabella, 144  
 Hoare, C. A. R., 85  
**HUGE\_VAL**, 268

**I**

Identificatore, 198  
**if-else**, ambiguità relativa a, 54, 235, 248

**if-else**, istruzione, 16, 18, 53, 235  
 illegale, aritmetica di puntatori, 101, 138, 214  
 implementazione, nascondere la, 75, 73, 75  
 implicita, dichiarazione di funzione, 23, 70, 209  
 incoerente, dichiarazione di tipo, 70  
 incompleto, tipo, 222  
 incremento, operatore di, ++, 14, 44, 45, 105, 212  
 indice di vettore, 19, 96, 209, 228  
 indici  
   di vettori e puntatori, 96, 97, 227  
   negativi di vettori, 98  
 indirezione, operatore di, \*, 92, 212  
 indirizzo  
   di registro, 221  
   di variabile, 24, 92, 94  
 infinito, ciclo, **for(;;)**, 58-59, 87  
 ingresso e uscita di dati *si veda*  
   input/output  
 inizializzatore, 240  
   forma di, 83, 218  
 inizializzazione, 38, 83, 230  
   all'interno di un blocco, 82, 235  
   default, 84, 230  
   di puntatore, 100, 138  
   di struttura, 129, 230  
   di un'unione, 231  
   di variabili automatiche, 28, 38, 83, 230  
   di variabili esterne, 38, 79, 83, 230  
   di variabili statiche, 38, 83, 230  
   di vettore bidimensionale, 110, 231  
   di vettore, 84, 111, 230  
   di vettori di strutture, 133  
   tramite costante stringa, 83-84, 230  
 inode, 184  
**input**  
   con buffer, 175  
   da tastiera, 12, 154, 174  
 formattato, *si veda* **scanf**  
 restituzione di un carattere, 76  
   senza uso di buffer, 175  
 input/output  
   di caratteri, 12, 154  
   errori relativi a, 163, 266  
   modello di, 12  
   redirezione di, 154, 155, 172  
**install**, funzione, 145

**L**

**labs**, funzione della libreria, 271  
**ldexp**, funzione della libreria, 269  
**ldiv**, funzione della libreria, 271  
 lessicale, campo di visibilità, 241  
 lessicali, convenzioni, 197-198  
 lessicografico, ordinamento, 117  
 libreria, funzione della, 3, 66, 69, 77, 90  
 linkage, 201, 241-242  
   esterno, 72, 198, 202, 240, 241  
   interno, 202, 240  
 lista di directory, programma per la lista di, 184  
**localtime**, funzione della libreria, 275

logica  
operatore di negazione !, 40, 213  
valore numerico di espressione, 40  
logico  
operatore AND, &&, 18, 40, 47, 216  
operatore OR, ||, 18, 40, 47, 217  
`long`, costante, 35, 200  
`long`, tipo 6, 15, 35, 203, 221  
`long double`, costante, 35, 200  
`long double`, tipo, 35, 203  
`LONG_MAX`, `LONG_MIN`, 270  
`longjmp`, funzione della libreria, 273  
`lookup`, funzione, 145  
`lower`, funzione, 41  
`ls`, comando, 184  
`lseek`, chiamata di sistema, 178  
lunghezza  
di nomi, 34, 198  
di nomi di variabile, 198  
di stringhe, 27, 36-37, 102  
lunghezza massima di riga, programma, 25, 28-29  
`lvalue`, 204

**M**  
macro con argomenti, 87  
macro, preprocessore di, 86, 242  
maggiore  
di, operatore di, >, 39, 215  
o uguale di, operatore di, >=, 39, 215  
magici, numeri, 11  
`main`, funzione, 3  
`main`, `return` da, 23, 160  
`makepoint`, funzione, 130  
`malloc`, funzione, 193  
`malloc`, funzione della libreria, 143, 170, 270  
mancante  
specificatore del tipo, 221  
specificatore di classe di memorizzazione, 221  
manifesta, costante, 244  
membro di struttura, nome del, 129, 223  
`memchr`, funzione della libreria, 268  
`memcmp`, funzione della libreria, 268  
`memcpy`, funzione della libreria, 268  
`memmove`, funzione della libreria, 268  
memoria, allocatore di, 142, 190-195  
memoria, allocazione di, 220

**N**  
negativi, indici, 98  
`newline`, 198, 243  
carattere, \n, 3, 12, 16, 36, 200, 258  
nomi  
lunghezza di, 34, 198  
spazio di, 240  
normalizzazione nell'aritmetica di puntatori, 102, 206  
notazione polacca, 72  
`NULL`, 100  
nulla  
istruzione, 15, 235  
stringa, 38  
nullo  
carattere \0, 27, 38, 200  
puntatore, 100, 206

memorizzazione  
automatica, classe di, 28, 202  
classe di, 202  
dichiarazione di classe di, 220  
di vettori, ordine di, 110, 227-228  
specificatore classe di, 220  
auto, 220  
extern, 28, 30, 78, 220  
mancante, 221  
register, 81, 220  
static, 28, 81, 202  
static, 81, 220

`memset`, funzione della libreria, 268  
minore o uguale, operatore di, <=, 39, 215  
minuscolo, programma per la conversione in, 155  
`mktimes`, funzione della libreria, 274  
modalità d'accesso a file, 162, 258  
`modf`, funzione della libreria, 269  
moduli, programmazione strutturata in, 21, 25, 31, 65, 72-73, 106  
modulo, operatore di, %, 39, 213  
moltiplicativi, operatori, 213  
moltiplicazione, operatore di, \*, 39, 213  
`month_day`, funzione, 109  
`month_name`, funzione, 111  
`morecore`, funzione, 193  
multidimensionale, vettore, 109, 227  
multipli, compilazione di file, 68  
multiplo, assegnamento, 18  
mutuamente ricorsive, strutture, 140, 223

`numcmp`, funzione, 118, 119  
numeri, dimensioni di, 6, 15, 34, 276  
numerico  
valore di un'espressione logica, 40  
valore di un'espressione relazionale, 40  
nuovo stile, funzione, 210

**O**

`O_RDONLY`, `O_RDWR`, `O_WRONLY`, 176  
oggetto, 201, 212  
`open`, chiamata di sistema, 176  
`opendir`, funzione, 188  
operatore di indirizzo, &, 92, 212  
operatori  
additivi, 214  
aritmetici, 39  
associatività degli, 50, 207  
bit per bit, 46, 216  
di assegnamento, 40, 48, 217  
di scorrimento, 46, 214-215  
di uguaglianza, 39, 215-216  
diritto di precedenza degli, 14, 50, 93, 110, 116, 207  
moltiplicativi, 213  
relazionali, 13, 39, 215  
tabella degli, 51  
operazioni  
sulle unioni, 149  
 valide sui puntatori, 102  
ordinamento  
di righe di testo, 106, 117  
lessicografico, 117  
programma per, 106, 117  
ordine di traduzione, 241  
ordine di valutazione, 18, 47, 50-51, 61, 75, 88, 93, 207  
ottale  
costante carattere in notazione, 36  
costante, \ooo, 36, 200  
output  
formattazione dello *si veda printf*  
redirezione di, 154  
sullo schermo, 12, 154, 162, 164, 165  
overflow, 39, 208, 268, 273

**P**

parametro, 67, 87, 88, 97, 210  
definizione, 22, 209  
parentesi graffe, 3, 6, 53, 82, 84

parentesi, espressione tra, 208  
parole chiave  
elenco di, 198  
programma per il conteggio di, 132  
parole, programma per il conteggio di, 17-18, 139  
pattern, programma per la rilevazione di, 66, 67, 115  
permessi relativi ai file, 175, 177  
`perror`, funzione della libreria, 266  
pipe, 154, 165  
polacca, notazione, 72  
`pop`, funzione, 75  
portabilità XV, 41, 47, 147, 157, 171, 189  
posizione di parentesi, 7  
postfissi, ++ e -, 44, 45, 104  
potenze di, 21, 269  
`pow`, funzione della libreria, 21, 269  
`power`, funzione, 21, 23  
precedenze fra gli operatori, 14, 50, 93, 131, 207  
prefissi, ++ e -, 44, 45, 105  
preprocessore  
di macro, 86, 242  
nome del, `__FILE__`, 272  
nome del, `__LINE__`, 272  
nomi predefiniti del, 248  
operatore del, ##, 88, 244  
operatore del, #, 88, 244  
operatore del, `defined`, 89, 246  
primaria, espressione, 208  
`printf`, funzione, 85  
funzione della libreria 3, 7, 15, 154, 155, 260  
tabella degli esempi di, 9, 157  
tabella di conversioni di, 157, 261  
privato di segno, carattere, 42, 202  
programma  
calcolatrice, 70, 72, 74, 77, 160  
`cat`, 164, 165  
`dc1`, 121, 122  
`echo`, 113  
formato di, 7, 16, 20, 38, 138, 198  
`fsize`, 186  
leggibilità, 7, 50, 62, 84, 148  
per il conteggio  
degli spazi, 18, 57  
di caratteri, 15  
di parole, 16, 139

di parole chiave, 132  
di righe, 15-16  
per la concatenazione di file, 154, 162, 164  
per la conversione della temperatura, 4-5, 8-9, 11  
per la conversione in minuscolo, 155  
per la lista di directory, 184  
per la replica di file, 13, 175  
per la ricerca di pattern, 66, 67, 68, 114-115  
per la ricerca in tabella, 144  
per la riga più lunga, 25, 28-29  
per l'ordinamento, 106, 117  
`undcl`, 125  
programma, argomenti di *si veda* argomenti dalla riga di comando  
promozione degli argomenti, 43, 210  
promozione integrale, 42, 204  
prototipo di funzione, 23, 43, 44, 70, 86, 118, 210  
provvisoria, definizione, 240  
`ptintrect`, funzione, 131  
`ptrdiff_t`, nome di tipo, 101, 148, 214  
puntatore  
    argomento, 95  
    void \*, 92, 102, 118, 207  
puntatore-intero, conversione, 206, 213  
puntatori  
    a file, 163, 179, 258  
    a funzione, 117, 147, 209  
    a strutture, 137  
    aritmetica di, 97, 99-102, 138, 213-214  
    aritmetica illegale di, 101, 138, 214  
    confronti fra, 100, 138, 193, 215-216  
    conversioni di, 206, 213  
    dichiarazioni di, 92, 94, 96, 99, 227  
    e indici, 96, 97, 228  
    generazione di, 208  
    inizializzazione di, 100, 138  
    normalizzazione nell'aritmetica di, 102, 206  
    nullo, 113, 206  
    operazioni permesse sui, 102  
    sottrazione fra, 102, 138, 206  
    vettori di, 106  
    vs. vettori, 96, 96-98, 103, 111  
    punto e virgola, 6, 10, 11, 15, 53, 55  
`push`, funzione, 75

`putc`, funzione della libreria, 163, 264  
`putc`, macro, 181  
`putchar`, funzione della libreria, 12, 154, 163, 264  
`puts`, funzione della libreria, 167, 264

## Q

`qsort`, funzione della libreria, 271  
`qsort`, funzione, 85, 107, 108, 118  
qualificatore di tipo, 217, 221  
`quicksort`, 85, 108

## R

`raise`, funzione della libreria, 274  
`rand`, funzione della libreria, 270  
`rand`, funzione, 44  
`RAND_MAX`, 270  
`read`, chiamata di sistema, 174  
`readdir`, funzione, 188, 189  
`readlines`, funzione, 107, 109  
`realloc`, funzione della libreria, 270  
redirezione *si veda* input/output,  
    redirezione dello  
`register`, specificatore di classe di memorizzazione, 81, 220  
registro, indirizzo di, 221  
relazionale  
    operatori, 13, 39, 215  
    valore numerico di espressione, 40  
`remove`, funzione della libreria, 259  
`rename`, funzione della libreria, 259  
restituzione all'input, 75  
`return`  
    conversione di tipi tramite, 71, 237  
    da `main`, 23, 160  
    istruzione, 22, 26, 68, 71, 237  
`reverse`, funzione, 60  
`rewind`, funzione della libreria, 265  
Richards, M., XIII  
ricorsione, 84, 139, 142, 187, 210, 0269  
ricorsivo, analizzatore sintattico discendente, 122  
rentri, 7, 16, 20, 54, 55  
righe  
    congiunzione di, 243  
    programma per il conteggio di, 15-16  
rimozione di definizione *si veda* `#undef`  
Ritchie, D. M., XI

## S

salto pagina, carattere, \f, 36, 200  
salto, istruzioni, 237  
`sbrk`, chiamata di sistema, 193  
`scanf`  
    funzione della libreria, 95, 155, 264  
    soppressione dell'assegnamento in, 160, 262  
tabella di conversioni di, 161, 263  
scelta, istruzione di, 235  
scelta fra più opzioni, 19-20, 55  
schermo, output su, 12, 154, 162, 164, 165  
scientifica, notazione, 71  
scorrimento, operatori di, 46, 214-215  
`SEEK_CUR`, `SEEK_END`, `SEEK_SET`, 265  
segno  
    carattere con, 42, 202  
    estensione del, 42, 175, 200  
senza buffer  
    `getchar`, 175  
    `input`, 175  
separata, compilazione, 65, 78, 80, 240  
sequenze di controllo, 4, 16, 18, 200, 242  
    \|hh esadecimale, 36, 200  
elenco di, 36, 200  
sequenziale, esecuzione di istruzioni, 234  
`setbuf`, funzione della libreria, 260  
`setjmp`, funzione della libreria, 273  
`setvbuf`, funzione della libreria, 260  
Shell, D. L., 60  
`shellsort`, funzione, 60  
short, tipo, 6, 34, 202, 221  
`SIG_DFL`, `SIG_ERR`, `SIG_IGN`, 273  
`signal`, funzione della libreria, 273  
simboli di costante, lunghezza di, 34, 35  
sin, funzione della libreria, 268  
sinh, funzione della libreria, 269  
sinistra, operatore di scorrimento a, <<, 47, 214-215  
sintassi di nomi di variabile, 34  
sintattica, albero dell'analisi, 122  
sintattica, notazione, 201  
sintattico, analizzatore ricorsivo discendente, 122  
sistema, chiamate di, 174  
`size_t`, nome di tipo, 102, 135, 148, 213, 258  
`sizeof`, operatore, 89, 102, 135, 211, 258  
sottolineatura (underscore), carattere di, 34, 198, 258  
sottovettore, argomento, 98

**strcpy**, funzione della libreria, 267  
**strcpy**, funzione, 103-104  
**strcspn**, funzione della libreria, 267  
**strup**, funzione, 143  
**strerror**, funzione della libreria, 267  
**strftime**, funzione della libreria, 275  
**strindex**, funzione, 66  
**stringa**  
  costante, 3, 16, 20, 27, 36, 98, 102, 200  
  costante estesa, 200  
  inizializzazione tramite costante, 83-84, 230  
  letterale *si veda*, stringa, costante  
  lunghezza di, 27, 37, 101, 102  
  tipo di, 208  
**stringhe**, concatenazione di, 37, 88, 201  
**strlen**, funzione della libreria, 267  
**strlen**, funzione, 37, 98, 101  
**strncat**, funzione della libreria, 267  
**strcmp**, funzione della libreria, 267  
**strncpy**, funzione della libreria, 267  
**strpbrk**, funzione della libreria, 267  
 **strrchr**, funzione della libreria, 267  
**strspn**, funzione della libreria, 267  
**strstr**, funzione della libreria, 267  
**strtod**, funzione della libreria, 269  
**strtok**, funzione della libreria, 267  
**strtol**, **strtoul**, funzioni della libreria, 270  
**struct**, specificatore, 222  
**struttura**  
  autoreferenziale, 139, 223  
  contrassegno di, 128, 222  
  dichiarazione di, 128, 221  
  dimensione di, 138, 213  
  inizializzazione di, 129, 231  
  nome del membro di, 129, 223  
  puntatori a, 137  
**strutture**  
  annidate, 129  
  inizializzazione di vettori di, 133  
  mutuamente ricorsive, 140, 223  
  operatore membro di, 129, 211  
  operatore puntatore alle, ->, 132, 211  
  semantica di riferimenti a, 211  
  sintassi di riferimenti a, 211  
  vettori di, 132  
**suffisso .h** del nome di un file, 30  
**suffisso di costante**, 200  
**swap**, funzione, 85, 93, 94, 108, 119  
**switch**, istruzione, 56, 73, 235

**syscalls.h**, intestazione, 175  
**system**, funzione della libreria, 169, 271

**T**  
**tabella**  
  di conversioni di **printf**, 157, 261  
  di conversioni di **scanf**, 161, 263  
  di esempi inerenti a **printf**, 9, 157  
  di operatori, 51  
  programma per la ricerca in, 144  
**malloc**, funzione, 142  
**tan**, funzione della libreria, 268  
**tanh**, funzione della libreria, 269  
**tastiera**, input da, 12, 154, 174  
**temperatura**, programma di conversione  
  della, 4-5, 8-9, 11  
**terminazione** di programmi, 165  
**testo**, flusso di, 12, 153, 258  
**testo**, ordinamento di righe di, 106, 117  
Thompson, K. L., XIII  
**time**, funzione della libreria, 274  
**time\_t**, nome di tipo, 274  
**tipi** di dati fondamentali, 6, 34, 202  
**tipi integrali**, 203  
**tipi**  
  aritmetica di, 203  
  derivati, XIII, 6, 203  
  equivalenza di, 233  
  virgola mobile, 203  
**tipo**  
  conversione dovuta a **return** del, 71, 237  
  di costante, 35, 200  
  di stringa, 208  
  dichiarazione di, 226  
  dichiarazione incoerente di, 69, 71  
  incompleto, 221  
  nomi di, 232  
  operatore di conversione di *si veda* cast  
  qualificatore di, 217, 221  
  regole di conversione di, 40, 42, 205  
  specificatore del, 221  
    mancante del, 221  
**TMP\_MAX**, 259  
**tmpfile**, funzione della libreria, 259  
**tmpnam**, funzione della libreria, 259  
**token**, 198, 243  
  concatenazione di, 244  
  sostituzione di, 243  
**tolower**, funzione della libreria, 155, 169, 267

**toupper**, funzione della libreria, 169, 267  
**traduzione**  
  ordine di, 241  
  unità di, 197, 241  
**treeprint**, funzione, 141, 142  
**trim**, funzione, 62  
**triplice**, sequenza, 242-243  
**troncamento**  
  di numeri a virgola mobile, 43, 204-205  
  dovuto a divisione, 7, 39, 214  
**typedef**, dichiarazione, 146, 221, 233  
**types.h**, intestazione, 186, 188

**U**  
**uguaglianza**, operatore di, ==, 16, 39, 215  
**ULONG\_MAX**, 270  
**unario**  
  operatore meno, -, 212  
  operatore più, +, 212  
**undcl**, programma, 125  
**underflow**, 39, 268  
**ungetc**, funzione della libreria, 169, 265  
**ungetch**, funzione, 76  
**union**  
  allineamento tramite, 191  
  dichiarazione, 148, 222  
  specificatore, 222  
**unione**  
  contrassegno (tag) di, 222  
  inizializzazione di, 231  
**unioni**, operazioni sulle, 149  
**unità di traduzione**, 197, 241  
**UNIX**, file system di, 174, 177, 184  
**unlink**, chiamata di sistema, 176, 178  
**uno**, operatore di complemento a, ~, 47, 212  
**unsigned char**, tipo, 35, 175, 181  
**unsigned long**, costante, 35, 199, 246  
**unsigned**  
  costante, 35, 200  
  tipo, 35, 49, 202, 221

**V**  
**va\_list**, **va\_start**, **va\_arg**, **va\_end**, 158, 178, 262, 272  
**valutazione**, ordine di, 18, 47, 50-51, 61, 75, 88, 93, 207  
**variabile**, 201  
  automatica, 28, 72, 81, 82, 202

**esterna**, 28, 71, 202  
**indirizzo di**, 24, 92  
**lista di argomenti di lunghezza**, 157, 178, 210, 229, 238, 272  
**lunghezza di nomi di**, 198  
**sintassi di nomi di**, 34  
**vecchio stile**, funzione, 23, 30, 71, 210  
**verticale**, carattere di tabulazione, \v, 36, 198, 200  
**vettore**  
  accesso tramite indici a, 19, 96, 215, 228  
  bidimensionale, 109, 110, 231  
    inizializzazione di, 110, 231  
  come argomento, 24, 26, 98, 110  
  conversione del nome, 97, 207  
  di caratteri, 20, 25, 102  
  di puntatori, 106  
  di strutture, 132  
  dichiarazione di, 19, 109, 227  
  dimensione di default, 84, 112, 134  
  inizializzazione di, 83-84, 111, 230  
  multidimensionale, 109, 227  
  ordine di memorizzazione di, 110, 227  
  riferimenti a, 209  
  vs puntatore, 96-98, 103, 111  
**vincoli di allineamento**, 138, 142, 149, 191, 206  
**virgola mobile**  
  costante, 9, 35, 200  
  intero, conversione, 43, 204  
  tipi a, 203  
    troncamento di quantità a, 43, 204  
**virgola**, operatore, 60, 218  
**virgolette**, ", 3, 4, 11, 16, 31, 36, 200  
**visibilità** *si veda* campo di visibilità  
**visibilità**  
  di etichette, 64, 234, 241  
  di variabili automatiche, 78, 241  
  di variabili esterne, 78, 241  
    regole inerenti a, 78, 240  
**visualizzabili**, caratteri, 266  
**void \***, puntatore, 92, 102, 118, 207  
**void**, lista degli argomenti, 30, 71, 229, 238  
**void**, tipo, 27, 203, 207, 220  
**volatile**, qualificatore, 203, 221  
**vprintf**, **vfprintf**, **vsprintf**, funzioni  
  della libreria, 178, 262  
**vuota**, istruzione *si veda* nulla, istruzione  
**vuota**, stringa, 36

**W**

wchar\_t, nome di tipo, 200  
while vs. for, 11, 58  
while, istruzione, 6–7, 58, 237  
write, chiamata di sistema, 175  
writelines, funzione, 107, 108

**Z**

zero, omissione della condizione pari a, 54,  
104