DisPenseOnline
Università degli Studi di Pavia
Biblioteca Delle Scienze

PaviaUniversityPress
Edizioni dell'Università degli Studi di Pavia

Tullio Facchinetti, Cristiana Larizza e Alessandro Rubini

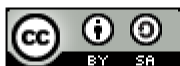
Dalla A alla Z
passando per C

Biblioteca Delle Scienze
Università degli Studi di Pavia
2009

Dalla A alla Z passando per C / Tullio Facchinetti,
Cristiana Larizza e Alessandro Rubini. - Pavia :
Biblioteca Delle Scienze, 2009. - 234 p. ; 24 cm . -
(Dispense Online)

Soggetto: Elaboratori elettronici. Linguaggio C.
Classificazione: 005.133

© Tullio Facchinetti, Cristiana Larizza, Alessandro Rubini – Pavia, 2009
ISBN: 978-88-903824-3-7


Questo testo è liberamente scaricabile dal sito http://www.dispenseonline.net.
La versione cartacea e la versione elettronica sono distribuite sotto licenza Creative Commons Attribution Share-Alike 3.0. Per maggiori informazioni si veda la pagina web http://creativecommons.org/licenses/by-sa/3.0/.

www.dispenseonline.net
www.paviauniversitypress.it/didattica

Pubblicato da:

Biblioteca Delle Scienze
Università degli Studi di Pavia
Via Bassi, 6
27100 Pavia
www.unipv.it/bibscienze

Grafica e stampa:

Print Service
Strada Nuova, 67
27100 Pavia

Sommario

Introduzione xiii
I.1 Organizzazione del documento xiii
I.2 Note sugli argomenti trattati xv

1 Ambiente di programmazione 1
1.1 L'autenticazione 1
1.2 L'interprete dei comandi: la shell 2
1.3 Sintassi dei comandi UNIX 3
1.4 Il *file system* 4
1.5 Scrittura, redirezione e visualizzazione di file 4
1.6 Le directory 5
1.6.1 Percorsi relativi e assoluti 5
1.6.2 Visualizzazione della directory corrente 6
1.6.3 La home directory 7
1.6.4 Le directory . e .. 7
1.6.5 Spostarsi tra directory 7
1.6.6 Elencare il contenuto di una directory 8
1.6.7 Creare e cancellare directory 9
1.6.8 Copiare file: cp 10
1.6.9 Spostare i file: `mv` 11
1.6.10 Cancellare file: `rm` 12
1.7 Le wildcard 13

1.8 Visualizzazione del manuale 14
1.9 Esecuzione di comandi 14

2 Dal problema al programma, passando dall'algoritmo 17

2.1 Esempi di problemi 18
2.2 L'algoritmo 18
2.3 Proprietà di un algoritmo 19
2.4 Esempi di algoritmo 19
2.4.1 Prodotto di matrici 20
2.4.2 Calcolo del massimo comune divisore 20
2.5 La programmazione 20
2.6 Il programma 20
2.7 Correttezza di un programma 21
2.8 Esecuzione del programma 21
2.9 Errori e debug 21
2.10 Testing 22
2.11 Manutenzione 22
2.12 Linguaggi di programmazione 22
2.13 Evoluzione del linguaggio C 23

3 Realizzazione di un programma 25

3.1 Il file sorgente 26
3.2 Il compilatore 26
3.3 Il linker e i file oggetto 26
3.4 Il loader 27
3.5 Strutturazione del codice 27

4 Concetti di base 29

4.1 Il primo programma in C 30
4.2 Compilazione del programma 31
4.2.1 Errori ed interpretazione dei messaggi del compilatore 32
4.3 I commenti 34
4.4 Nota stilistica 34
4.5 Gli identificatori 34
4.6 Le parole chiave 35
4.7 Le variabili 36
4.8 Visualizzazione a video 37
4.9 Lettura di dati da tastiera 37
4.10 Specifica di formato per `printf` e `scanf` 38

5 Istruzioni e strutture di controllo 41

5.1 Istruzioni composte 42

5.1.1 L'operatore virgola 43
5.2 Il costrutto `while` 43
5.3 Il costrutto `do-while` 44
5.4 Il costrutto `for` 45
5.5 Il costrutto `if` 48
5.6 Il costrutto `switch` 49
5.7 Le istruzioni `break` e `continue` 51
5.8 Il costrutto `goto` 52

6 Tipi di dati 55

6.1 Tipi di dato e memoria 56
6.2 Tipi interi 56
6.2.1 Costanti di tipo intero 57
6.3 Tipi a virgola mobile 57
6.4 I puntatori 59
6.5 Gli array 60
6.5.1 I vettori 60
6.5.2 Uso delle macro in combinazione con vettori e cicli 62
6.5.3 Le matrici 62
6.5.4 Array multi-dimensionali 64
6.5.5 Memorizzazione degli array 64
6.6 Strutture dati 64
6.7 Union 66
6.8 Interi indipendenti dalla piattaforma 66
6.9 Conversioni di tipo 67
6.10 Assegnare nuovi nomi ai tipi di dato: `typedef` 68
6.10.1 `typedef` vs. `#define` per definite nuovi tipi 69

7 I puntatori 71

7.1 Puntatori a `void` 72
7.2 Puntatori e vettori 72
7.3 L'operatore `sizeof` 75
7.4 Le stringhe 76
7.4.1 Dettagli sull'inizializzazione 78
7.4.2 Esempio di funzione per la manipolazione delle stringhe 79
7.5 Vettori di puntatori/stringhe 79
7.6 Puntatori e strutture 80
7.6.1 Puntatori a strutture come campi di strutture 81
7.7 Allocazione dinamica della memoria 82

8 Funzioni 87

8.1 Dichiarazione di funzioni 88
8.2 Definizione di funzioni 88

8.3 Strutturazione del programma in funzioni 89
8.4 Passaggio dei parametri 90
8.5 Passaggio per riferimento 91
8.6 La funzione `main` 92
8.7 Puntatori a funzione 93
8.8 Numero variabile di parametri 96

9 Gli operatori 97

9.1 Precedenza degli operatori 98
9.2 Ordine di valutazione 98
9.3 Il concetto di side effect 98
9.4 Side effect e ordine di valutazione 98
9.5 Associatività degli operatori 99
9.6 Chiamata a funzione 100
9.7 Elemento di vettore 100
9.8 Elemento di struttura 101
9.9 Elemento di struttura da puntatore 101
9.10 Negazione logica 102
9.11 Complemento a 1 103
9.12 Negazione unaria 103
9.13 Incremento e decremento 104
9.14 Estrazione di un indirizzo 105
9.15 Uso di puntatore 105
9.16 Operatore di casting 106
9.17 Dimensione di una variabile 106
9.18 Moltiplicazione e divisione 107
9.19 Resto di divisione intera 108
9.20 Somma e sottrazione 109
9.21 Spostamento dei bit (shift) 110
9.22 Confronto 111
9.23 Confronto: uguaglianza e diversità 112
9.24 AND bit-a-bit 113
9.25 XOR bit-a-bit 114
9.26 OR bit-a-bit 114
9.27 AND logico 115
9.28 OR logico 116
9.29 Espressione condizionale 116
9.30 Assegnamento 117
9.31 Forme abbreviate di assegnamento 118
9.32 Operatore virgola 119

10 Classi di memoria 121

10.1 La visibilità di dati e funzioni 121

10.2 Classi di memorizzazione 123
10.3 Allocazione di memoria 124
10.4 Variabili auto 126
10.5 Variabili `register` 126
10.6 Variabili `static` 127
10.7 Variabili extern 128
10.8 Inizializzazioni 129

11 Il preprocessore 131

11.1 La direttiva #define 132
11.2 La direttiva #include 132
11.3 La direttiva #if e #ifdef 133

12 I file 137

12.1 File binari e file di testo 137
12.2 Accesso a file 138
12.3 Apertura e chiusura di file 139
12.3.1 Apertura di file 139
12.3.2 Chiusura di file 140
12.4 Forzare la scrittura dei dati con `fflush` 140
12.5 Accesso a file binari: le funzioni `fread` e `fwrite` 141
12.6 Lettura di file di testo 142
12.7 Redirezione dell'input e dell'output 144
12.7.1 Il programma `count.c` 144
12.7.2 Un altro esempio di redirezione da linea di comando 146
12.8 Lettura di file strutturati con `fgets` e `sscanf` 147
12.9 I/O con le funzioni `fscanf` e `fprintf` 147
12.10 Esempio di lettura di matrici con `fgets` e `sscanf` 148
12.11 `fscanf` e `fgets` a confronto 150
12.11.1 La bufferizzazione dell'input 150
12.11.2 Il problema del buffer overflow 152

13 Programmi composti da più file sorgente 155

13.1 Esempio di utilizzo di più file sorgente 155
13.2 Il comando `make` 158
13.3 Esempio di dipendenze 159
13.4 Il makefile 159

14 Le librerie 161

14.1 Uso di librerie esterne 161
14.1.1 Il comando `ar` 163
14.1.2 Esempio di utilizzo del comando `ar` 163

14.2 La libreria di input/output `stdio` 164
14.2.1 Printf 164
14.3 La libreria standard `stdlib` 165
14.3.1 Controllo dell'esecuzione del programma 165
14.3.2 Gestione della memoria 166
14.3.3 Generazione di numeri casuali 166
14.4 Manipolazione di stringhe 168
14.5 La libreria matematica 169

15 Stile di programmazione 171

15.1 Indentazione 172
15.2 Indentazione ed errori 173
15.3 Indicazioni varie 174

16 Tecniche di programmazione 177

16.1 La ricorsione 177

17 Strutture informative 181

17.1 Classificazione delle strutture di dati 181
17.2 Strutture astratte di dati 182
17.3 La lista lineare 183
17.4 La coda 184
17.5 La pila (stack) 184
17.6 La doppia coda 189
17.7 Gli array 189
17.8 Le tavole (tabelle) 190
17.9 I grafi 190
17.10 Gli alberi 192
17.10.1 Visita degli alberi 193
17.10.2 Visita in ordine anticipato 193
17.10.3 Visita in ordine differito 194
17.11 Alberi binari 195
17.11.1 Visita in ordine simmetrico 195

18 Strutture concrete di dati 197

18.1 Struttura sequenziale 197
18.2 Catena o lista 200
18.2.1 Inserimento ed eliminazione di elementi 201
18.3 Memorizzazione delle strutture astratte: liste 203
18.4 Memorizzazione delle strutture astratte: code, pile, doppie code 203
18.5 Memorizzazione delle strutture astratte: matrici 203
18.6 Memorizzazione delle strutture astratte: tavole 204

18.7 Ricerca sequenziale 204
18.8 Ricerca binaria 204
18.9 Accesso diretto 204
18.10 Accesso calcolato (hashing) 205
18.11 Memorizzazione di alberi e grafi in catene 205
18.12 Memorizzazione di alberi e grafi in plessi 205

19 Tecniche di programmazione e algoritmi base 207

19.1 L'ordinamento 207
19.1.1 Bubblesort 207
19.2 Algoritmi di ricerca 209
19.2.1 La ricerca sequenziale 209
19.2.2 La ricerca binaria 210
19.2.3 Hashing 212
19.3 Ricerca e ordinamento con le funzion di libreria 215

20 Esercizi e algoritmi 219

20.1 Programmazione in C 219
20.2 Realizzazione di algoritmi 227

Bibliografia 229

Appendice A: Tabella degli operatori 231

Appendice B: Il compilatore gcc 233
B.1 Opzioni più importanti 233

Introduzione

Questo testo viene utilizzato per la parte relativa alla programmazione in Linguaggio C, nel contesto del corso di Fondamenti di Informatica presso l'Università di Pavia e di Mantova, tenuto dal Prof. Tullio Facchinetti.

Parte del contenuto è una rielaborazione delle dispense redatte dal Prof. Alessandro Rubini per il corso di Sistemi Real-time presso l'Università di Pavia e disponibile online [2, 3]. Se il documento del Prof. Rubini è "dalla A alla X", questo si propone di essere "dalla A alla Z", quindi una sostanziale estensione della versione originale del Prof. Rubini.

Vari spunti sono tratti e adattati dal materiale utilizzato dal Prof. Luca Lombardi, mentre la maggior parte degli esercizi derivano dalle prove d'esame realizzate in passato dalla Prof.ssa Cristiana Larizza.

I.1 ORGANIZZAZIONE DEL DOCUMENTO

Il presente documento è organizzato come segue.

- Il capitolo 1 introduce l'ambiente di programmazione basato sulla shell di UNIX; Linux viene tipicamente utilizzato per gli esempi e gli esercizi. Viene fatta una panoramica dei comandi base della shell, che possono quindi essere utilizzati per gestire i file sorgente e i programmi compilati che verranno realizzati nel corso delle esercitazioni.

- Nel capitolo 2 sono presentati i concetti relativi agli algoritmi e le loro proprietà; viene introdotta la nozione di programma, con le relative caratteristiche, come mezzo informatico per la risoluzione di problemi.

- Nel capitolo 3 vengono discussi i passaggi e i componenti necessari per la realizzazione di un programma; gli aspetti trattati sono validi per il linguaggio C, ma possono essere applicati anche a vari altri linguaggi di programmazione.

- I concetti di base del linguaggio C vengono proposti nel capitolo 4, nel quale si presenta il più semplice programma in C, e vengono discussi gli identificatori, i commenti, le parole chiave, le variabili e introdotto l'input/output per l'interazione con l'utente.

- Le strutture di controllo sono introdotte nel capitolo 5, che tratta i costrutti condizionali e quelli per realizzare cicli.

- Il capitolo 6 descrive la tipizzazione del C, presentando tutti i tipi di dati disponibili: intero, virgola mobile, puntatore, struttura dati e le `union`, che sono brevemente accennate. Nel contesto del capitolo sono inoltre presentati gli array, sia mono-dimensionali che multi-dimensionali, le tipologie di conversione di tipo e l'utilizzo del comando `typedef` per definire nuovi tipi di dato.

- Vista la delicatezza e l'importanza dell'argomento, un intero capitolo, il capitolo 7, è dedicato alla trattazione dei puntatori. Ne viene analizzato l'utilizzo in combinazione con vettori e strutture dati, e vengono trattate nel dettagli le stringhe. È introdotto l'operatore `sizeof`. Viene descritto il meccanismo di passaggio di parametri da linea di comando a un programma e sono accennati i puntatori a funzione.

- Nel capitolo 8 viene illustrata la suddivisione in funzioni di un programma, per mezzo di dichiarazione e definizione di funzioni, e passaggio dei parametri. Vengono riprese le caratteristiche della funzione `main` e del passaggio di parametri da linea di comando al programma.

- Tutti gli operatori del linguaggio C sono presentati nel capitolo 9: aritmetici, logici, i cosiddetti operatori *bit-a-bit*, ecc. Sono inoltre introdotti i concetti di precedenza, ordine di valutazione, *side effect*, associatività e *lvalue*.

- Il capitolo 10 presenta tutte le problematiche relative alle classi di memoria di funzioni e variabili, la loro visibilità, il problema e le differenze nelle inizializzazioni, e le funzioni per l'allocazione dinamica della memoria.

- Il tema del preprocessore viene illustrato nel capitolo 11, dove sono spiegate tutte le direttive per il preprocessore, con particolare attenzione alla definizioe di macro e all'inclusione di file di intestazione.

- Nel capitolo 12 viene trattato il problema dell'accesso ai file, con distinzione tra file binari e di testo. Viene presentata la tecnica di redirezione dell'input, e vengono analizzate le differenze, presentando vantaggi e svantaggi, delle varie funzioni disponibili per l'input/output da file.

- La creazione di programmi costituiti da più file sorgente è oggetto del capitolo 13, che comprende una breve introduzione all'utilizzo del comando `make`.

- Il capitolo 14 tratta le librerie e le funzioni delle librerie standard. Vengono illustrate la libreria matematica, quella di I/O, e di manipolazione delle stringhe.

- Un accenno allo stile di scrittura di programmi viene fatto nel capitolo 15.

- Nel capitolo 16 vengono trattate tecniche di programmazione e strutture dati di alto livello tipicamente utilizzate in programmi e algoritmi complessi. Viene trattata la tecnica della ricorsione, le liste, la pila e le code, e diversi altre. Al momento questa sezione tratta le strutture informative dal punto di vista per lo più teorico, ma si prevede entro breve di aggiungere opportuni esempi in linguaggio C che le implementino.

- Alcuni spunti per esercitazioni sono forniti nel capitolo 20.

- Nell'appendice A viene fatto un riassunto degli operatori, ordinato a seconda della precedenza, e vengono riportate le loro caratteristiche essenziali.
- Infine, un accenno al compilatore `gcc` viene presentato nell'appendice B.

I.2 NOTE SUGLI ARGOMENTI TRATTATI

La versione attuale di questo documento non tratta approfonditamente alcuni argomenti che fanno parte dello standard del Linguaggio C, ma che sono ritenuti non particolarmente interessanti ai fini didattici per l'apprendimento del linguaggio. È però opportuno menzionare quantomento tali argomenti, in modo che il lettore interessato possa documentarsi su di essi attingendo da altre fonti.

I temi che sono trattati solo marginalmente, o che non vengono proprio trattati in questo documento sono i seguenti:

- Le `union` sono accennate al momento di illustrare le strutture di dati, ma il loro utilizzo non viene approfondito, dal momento che costituiscono in tipo di dato che viene utilizzato in contesti molto particolari;
- le enumerazioni non vengono trattate;
- i campi di bit non sono descritti.

Infine, la maggior parte delle caratteristiche che appartengono allo standard C99 non vengono considerate. Queste caratteristiche includono, tra le altre:

- il tipo di dato numerico complesso come tipo nativo;
- il tipo di dato nativo booleano `bool`;
- la possibilità di intercalare dichiarazioni di variabili e istruzioni;
- la possibilità di inizializzare gli array con un numero di elementi specificato da una variabile.

Capitolo 1

AMBIENTE DI PROGRAMMAZIONE

Per poter programmare in C sono necessari una serie di strumenti software che è meglio conoscere adeguatamente per garantire un'attività proficua. Tali strumenti, alcuni dei quali verranno ripresi e descritti più in dettaglio in seguito, sono ad esempio l'editor e il compilatore.

Molti ambienti per la programmazione in C forniscono ambienti di sviluppo cosiddetti "visuali", cioè che mettono a disposizione una interfaccia grafica dalla quale è possibile controllare il processo di realizzazione del programma, ad esempio per mezzo di bottoni e altre facilitazioni. Per esempio, è possibile lanciare una compilazione senza preoccuparsi dei comandi che vengono effettivamente invocati per effettuare le operazioni richieste. Altri ambienti di programmazione, per contro, richiedono una maggiore conoscenza dei comandi e della gestione del computer.

In questo capitolo si illustreranno brevemente i concetti e i comandi da utilizzare in ambiente Unix per la gestione dei programmi, la compilazione, l'esecuzione di comandi accessori e altri aspetti di utilità generale.

1.1 L'AUTENTICAZIONE

All'accesso di un sistema Unix, come prima cosa verrà richiesto di effettuare la *login*, ovvero l'autenticazione. Dal momento che il sistema Unix supporta il collegamento di più utenti, contemporaneamente o in momenti diversi, è necessario poter individuare univocamente l'identità dell'utente che accede al sistema per poter gestire opportunamente le informazioni relative allo specifico utente.

In particolare, conoscere l'identità dell'utente che accede al sistema permette di gestire opportunamente i permessi di accesso a file e directory[1].

L'autenticazione avviene per mezzo della tipica accoppiata username/password che devono essere forniti al sistema. Il sistema rimane in attesa dell'autenticazione visualizzando l'apposita richiesta:

```
login:
```

Il processo di autenticazione inizia scrivendo il *nome utente*, detto anche *username* o *user id*. Lo username viene assegnato dall'amministratore del sistema, e con tale nome si viene univocamente identificati dal sistema stesso. Il sistema risponde chiedendo

```
Password:
```

A questo punto è possibile digitare i caratteri della password di accesso. Si noti che i caratteri battuti come password sono invisibili, cioè non vengono visualizzati sul video. Questo evita che la password possa essere "rubata" da malintenzionati eventualmente appostati alle spalle dell'utente che sta effettuando l'autenticazione.

Se la password è corretta, il sistema mostra il cosiddetto *prompt*. Un esempio di login effettuato con successo è il seguente (nel quale per chiarezza è visualizzata anche la password digitata):

```
login: user007
Password: pl67kf2
user007@europa:~$
```

Non ci preoccupiamo per ora del significato di ciò che viene visualizzato quando l'autenticazione ha successo. I dettagli a riguardo sono illustrati nel Capitolo 1.2.

Errori su username o password vengono opportunamente notificati, e la richiesta di login viene ripresentata finché l'autenticazione non ha successo. Per esempio:

```
login: utenet1
Password:
Login incorrect

login:
```

NOTA Viene fatta differenza tra caratteri maiuscoli e minuscoli. Quindi, per esempio, le tre parole `user007`, `USER007` e `User007` sono da considerarsi diverse.

1.2 L'INTERPRETE DEI COMANDI: LA SHELL

Nel corso della dispensa verranno talvolta illustrati i comandi che vengono impartiti al computer per effettuare operazioni necessarie per lo sviluppo di un programma, come la compilazione, l'esecuzione di programmi accessori, eccetera. Questo presuppone l'esistenza di un *interprete* che riceve i comandi e ne esegue le operazioni associate. L'interazione tra l'utente e l'interprete, per quanto ci interessa, avviene per mezzo della cosiddetta *linea di comando*, nella quale l'utente introduce i comandi sotto forma di *stringhe di testo*, ovvero sequenze di caratteri.

Il *prompt* è il segnale con cui il sistema si mostra pronto per altro input dopo aver elaborato l'input precedente (come la username o la password). Il prompt può essere personalizzato per mostrare informazioni quali lo username, il nome della macchina che si sta utilizzando (utile quando ci si

[1] I primi calcolatori che permettevano l'accesso a più di un utente usavano questa informazione anche per contabilizzare il tempo di utilizzo del calcolatore, che era una risorsa molto limitata, per farne poi pagare l'utilizzo.

collega a computer remoti), e la directory corrente. Negli esempi che verranno proposti si potrà trovare un prompt come il seguente:

```
user1@europa:~$
```

il quale può essere scomposto nei seguenti elementi:

```
user1   @   europa   :   ~   $
```

dove

- `user1` è lo username
- `@` è un carattere convenzionale (si legge "at") che introduce il nome del computer al quale si è collegati
- `europa` è il nome del computer in uso
- `:` è un carattere convenzionale che introduce il percorso della directory corrente (vedi Sezione 1.6)
- `~` è la directory corrente (vedi Sezione 1.6)
- `$` è un carattere convenzionale per delimitare il prompt dall'input dell'utente

Il prompt non viene proposto da Unix direttamente, ma da un programma chiamato *shell* che, come quelli che possono essere scritti dagli utenti, viene eseguito da Unix. La shell si occupa di interpretare i comandi inseriti dall'utente e di eseguire le azioni corrispondenti. È così chiamata in quanto costituisce lo strato più esterno del sistema Unix[2], cioè quello più vicino all'utente. Unix può infatti essere pensato come una serie di livelli logici posti tra la macchina e l'utente, il più esterno dei quali è appunto la shell. All'indirizzo [1] si può trovare una trattazione molto completa della shell.

I caratteri battuti sulla tastiera dall'utente sono riprodotti sullo schermo accanto al prompt, andando a formare la cosiddetta *riga di comando*.

La riga di comando viene presa in considerazione dalla shell solo quando viene premuto il tasto Invio/Enter/Return. Fino ad allora, il suo contenuto si trova in un'area provvisoria detta *buffer di input*, e può essere corretto con il tasto di cancellazione.

1.3 SINTASSI DEI COMANDI UNIX

La sintassi dei comandi UNIX prevede che le istruzioni siano specificate nel modo seguente

```
comando [opzioni] [argomenti]
```

Dove:

- `comando` indica l'operazione da compiere
- `[opzioni]` permette di specificare delle varianti al comando stesso
- `[argomenti]` specifica i parametri del comando (si tratta generalmente del nome dei file o delle directory su cui deve intervenire il comando stesso)

[2] In inglese, shell significa "conchiglia".

Le opzioni e gli argomenti sono racchiusi dalle parentesi quadre per indicare che sono dei parametri opzionali, cioè possono essere omessi.

Per esempio, il comando `man`, che verrà discusso in seguito più dettagliatamente, e che permette di consultare il manuale del comando specificato come argomento. I seguenti due comandi:

```
man cp
man man
```

che sono stati scritti senza visualizzare il prompt che li precede, visualizzano rispettivamente il manuale del comando `cp` e del comando `man` stesso (anche `man` è un comando!).

1.4 IL *FILE SYSTEM*

Il *file system* è un metodo per memorizzare e organizzare le informazioni in un sistema di calcolo [4]. I dati sono immagazzinati all'interno di *files*, che sono costituiti quindi dal blocco di informazioni che si intende memorizzare.

I file non contengono soltanto dati, ma possono contenere le istruzioni da eseguire corrispondenti ad un determinato programma (i cosiddetti programmi eseguibili).

> *NOTA* La maggior parte dei comandi illustrati in questo capitolo non sono altro che dei file nel file system che contengono le istruzioni corrispondenti al programma stesso. Il nome del comando corrisponde al nome del relativo file.

Ad ogni file è associato un nome, il quale può essere composto dai seguenti caratteri:

```
A...Z a...z 0...9 _ - . ,
```

In realtà cambiando sistema operativo cambia anche l'insieme dei caratteri validi per un nome di file. I caratteri sopra riportati sono però validi per pressoché ogni sistema operativo, e quindi in particolare anche per UNIX.

Si tenga presente che Unix distingue tra lettere maiuscole e minuscole. Alcuni esempi di nomi di file sono: `lezione.doc`, `LEZIONE.doc`, `Lezione.doc`, `lezione.old`, `file_mio`, `19.nov.92`; questi nomi di file rappresentano tutti file diversi tra loro.

1.5 SCRITTURA, REDIREZIONE E VISUALIZZAZIONE DI FILE

Un modo semplice per inserire nel file `agenda` i caratteri che formano la stringa `pranzo di lavoro` è:

```
user1@europa:~$ echo pranzo di lavoro > agenda
```

Il comando `echo` ripete (fa l'"eco") quello che gli si dice sulla riga di comando. In genere, tale ripetizione porta alla visualizzazione del contenuto della riga di comando sullo schermo. Per verificarlo, può essere istruttivo verificare quale è l'effetto del comando

```
user1@europa:~$ echo pranzo di lavoro
```

Il simbolo $>$ è l'*operatore di redirezione* che redirige l'output del comando precedente verso il file specificato di seguito. Normalmente i comandi inviano il loro output verso ciò che si chiama *standard output*, o `stdout`. Lo standard output è un file associato solitamente al terminale, quindi normalmente *scrivere sullo standard output* equivale a scrivere sul video.

Il risultato del comando nell'esempio è quello di redirigere lo standard output del comando `echo`, ovvero il testo che dovrebbe essere scritto sul terminale, cioè stampato a video, verso il file `agenda`. Se

il file `agenda` non esiste, allora viene creato, mentre se il file esiste già, allora esso viene sovrascritto. Nel secondo caso, il contenuto del file preesistente viene perduto. L'operatore di redirezione va quindi utilizzato con cautela, per evitare di eliminare dei dati importanti, che sarebbe poi impossibile recuperare.

Per vedere sullo schermo il contenuto del file `agenda` si utilizza il comando `cat`:

```
user1@europa:~$ cat agenda
pranzo di lavoro
```

Il comando `cat` sta per *con(cat)enate*, in quanto concatena tutti i file specificati sulla sua linea di comando verso il suo standard output. Per esempio

```
user1@europa:~$ cat file1 file2 file3
```

concatena il contenuto dei tre file `file1`, `file2` e `file3`, verso il suo standard output, anch'esso generalmente associato al terminale.

È possibile aggiungere (appendere) delle informazioni alla fine di un file esistente con il comando:

```
user1@europa:~$ echo cena fuori >> agenda
```

nel quale si utilizza l'operatore di redirezione formato dal "doppio maggiore" (>>) invece del comando di redirezione usuale.

Ora il file `agenda` avrà il seguente contenuto:

```
user1@europa:~$ cat agenda
pranzo di lavoro
cena fuori
```

infatti il file `agenda` esistente non è stato sovrascritto come nel caso in cui si fosse usato l'operatore >, ma il nuovo testo viene *accodato* al file esistente.

Ovviamente, anche lo standard output del comando `cat` può essere rediretto dal terminale verso un file. Per esempio, il comando

```
user1@europa:~$ cat agenda > copia_agenda
```

redirige l'output del comando `cat`, cioè il contenuto del file `agenda`, verso il file `copia_agenda`. Di fatto, in questo modo, è stata effettuata la copia del file `agenda`.

1.6 LE DIRECTORY

Le *directory* servono per organizzare in modo gerarchico i file, e costituiscono il file system. Ogni file ha una directory che lo contiene.

La struttura che deriva formalmente si chiama *grafo aciclico*, e in particolare assume le caratteristiche di un *albero*. Ogni directory ha un solo genitore, in cui è contenuta e di cui si dice figlia. La struttura delle directory si dice gerarchica perché la relazione genitore-figlio determina una gerarchia, come quella esemplificata in Figura 1.1. Di norma una qualsiasi directory (es. `user1`) si trova dentro un'altra directory. Ma esiste una directory che non è contenuta in nessun'altra: si chiama `root` (radice dell'albero) e si indica con la barra /. Questa barra si chiama *slash*, da non confondersi con la barra \, la quale è chiamata *backslash*.

1.6.1 Percorsi relativi e assoluti

Ad ogni istante è definita una directory corrente o di lavoro. File e directory, intesi come nodi di un albero, non possono essere individuati univocamente con un nome semplice, come per esempio

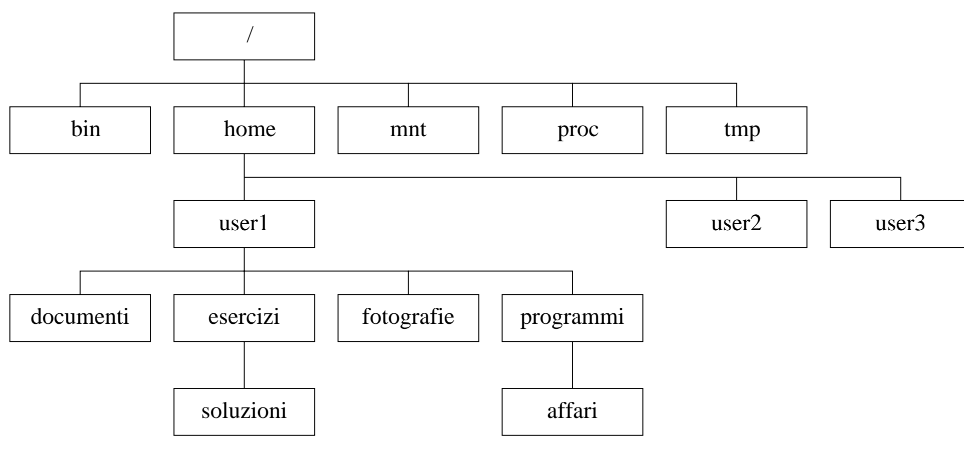


Figura 1.1 Esempio di albero delle directory.

`user1`, in quanto directory o file diversi possono avere lo stesso nome, qualora siano contenuti in directory diverse. Per questo ogni file o directory con nome semplice `fname` ha un nome completo o pathname che può avere due forme:

1. *assoluto*: localizza `fname` sull'albero *rispetto alla directory root*; è dato dai nodi da / fino a `fname` compreso, separati da /
   ad esempio: `/home/user1/esercizi/soluzioni`
2. *relativo*: localizza `fname` *rispetto alla directory corrente*; è dato dai nodi dalla directory corrente esclusa fino a `fname` compreso, separati da /
   ad esempio: `esercizi/soluzioni` è un percorso relativo alla directory corrente `/home/user1/`

Si notino le seguenti caratteristiche dei percorsi relativi/assoluti:

- un percorso si dice assoluto se inizia per /, altrimenti si dice relativo, quindi
- un percorso relativo NON inizia mai con lo slash;
- nella scrittura di un percorso, il carattere / ha 2 usi:
  1. indica la directory `root` se posto all'inizio come primo carattere del pathname
  2. funge da separatore di directory se posto in mezzo al pathname

1.6.2 Visualizzazione della directory corrente

Il comando `pwd` scrive sul suo standard output la directory corrente. Ad esempio, se ci si trova nella directory `programmi` di `user1`, si ottiene il seguente risultato:

```
user1@europa:~/programmi$ pwd
/home/user1/programmi
```

Il comando `pwd`, a differenza di altri comandi come `mkdir` e `rmdir`, non è implementato come un programma presente sul disco e chiamato `pwd`, ma è un comando interno alla shell[3].

[3]Questo perché la directory corrente è un attributo associato al processo corrente, ed è quindi diversa per ogni processo.

1.6.3 La home directory

A ciascun utente viene assegnata una directory nella quale è libero di effettuare tutte le operazioni di creazione, spostamento, modifica, cancellazione su directory e file. Tale directory viene generalmente creata all'interno di

```
/home
```

e viene detta *home directory*. L'associazione di un utente alla propria home directory è possibile grazie al processo di autenticazione. Per esempio la home directory dell'utente `user1` può essere la seguente:

```
/home/user1
```

Dal momento che la home directory è una directory speciale che viene usata come punto di riferimento per ciascun utente, il suo percorso viene anche abbreviato utilizzando il carattere ∼. Per ciascun utente, la directory ∼ corrisponde alla propria home directory. Il carattere ∼ può essere usato come sostitutivo del percorso assoluto di una directory. Per esempio, le seguenti forme sono del tutto valide:

```
~/esercizi/soluzioni
~/programmi
~/
```

e vengono interpretate, per l'utente `user1`, rispettivamente come

```
/home/user1/esercizi/soluzioni
/home/user1/programmi
/home/user1
```

1.6.4 Le directory . e ..

I nomi . e .. sono nomi speciali di directory:

- `.` rappresenta la directory corrente
- `..` rappresenta la directory genitore di quella corrente

Questi nomi di directory speciali possono essere utilizzati per formare dei percorsi sia relativi che assoluti. Per esempio, il seguente percorso assoluto:

```
/home/./user1/esercizi/././soluzioni/../../programmi/../fotografie
```

corrisponde, nel file system di esempio riportato in Figura 1.1, alla directory identificata dal percorso assoluto

```
/home/user1/fotografie
```

1.6.5 Spostarsi tra directory

Il comando `cd` serve a spostarsi tra le directory, cioè cambia sostanzialmente la directory corrente. Per esempio

```
user1@europa:~/programmi$ cd /home/user1/
user1@europa:~$ cd programmi
user1@europa:~/programmi$
```

Il comando `cd` è un comando interno alla shell.

Tenendo presente il concetto di percorso relativo e assoluto, è importante notare un errore tipico nella chiamata di un comando. Molto spesso infatti viene fatta confusione nello scrivere le giuste istruzioni per invocare il comando. Per esempio, il comando per spostarsi nella directory `programmi` è il seguente:

```
user1@europa: $ cd ./programmi
```

ed è ben diverso dal comando

```
user1@europa: $ ./programmi/cd
```

Infatti il secondo comando richiede al sistema di *ESEGUIRE* il programma che si chiama `cd` e che si trova nella sottodirectory `programmi` della directory corrente! Se tale comando non esiste, viene generato un messaggio di errore. Ma anche se esistesse, molto probabilmente non sarebbe il comando giusto per cambiare directory...
L'esempio è stato fatto per il comando `cd`, ma lo stesso tipo di errore *può essere commesso per qualsiasi comando.*

1.6.6 Elencare il contenuto di una directory

Il comando `ls` (list) mostra il contenuto della directory corrente. Il formato completo di `ls` è:

```
ls [opzioni] [lista di file o dir]
```

Le parentesi quadre indicano che il contenuto delle parentesi stesse è opzionale, cioè è possibile ometterlo nell'invocazione del comando senza pregiudicarne l'effetto. L'aggiunta di una o più opzioni può essere fatta per modificare leggermente il comportamento del comando. Le opzioni più importanti e utilizzate sono:

- `ls -a` elenca anche i file (normalmente invisibili) il cui nome comincia per `.` (punto)
- `ls -l` elenca in formato lungo
- `ls -t` elenca a partire dal file più recente
- `ls -C` visualizza l'elenco incolonnato
- `ls -R` (maiuscolo) elenca ricorsivamente anche le sotto-directory
- `ls -r` (minuscolo) elenca i file in ordine inverso sotto-directory
- `ls -f` non ordina i file, abilita l'opzione `-a`, disabilita la `-l` e i colori

Un esempio di utilizzo del comando `ls` è il seguente:

```
user1@europa:~$ ls -l .
-rw-r--r-- 1 toolleeo root 3500 2009-06-26 15:39 programma.c
```

nel quale si richiede di visualizzare il contenuto della directory corrente utilizzando il formato lungo, la quale contiene il solo file `programma.c`.

Il formato lungo visualizza una serie di utili informazioni riguardo ai file. Senza entrare nei dettagli del significato delle singole voci:

- `-rw-r--r--` specifica il tipo di file ed i permessi di accesso al file (si rimanda alla documentazione della shell per maggiori dettagli riguardo ai permessi)

- `1` indica il numero di cosiddetti "hard links" collegati al file (anche in questo caso, si rimanda il lettore a letture più approfondite per maggiori informazioni)
- `toolleeo` è l'utente che possiede il file
- `root` è il gruppo associato al file
- `3500` è la dimensione del file in byte
- `2009-06-26` e `15:39` sono la data e l'ora di ultima modifica del file
- `programma.c` è il nome del file

Un altro esempio è il seguente, che va interpretato tenendo presente l'albero delle directory di Figura 1.1 e il fatto di essere posizionati inizialmente all'interno della home directory di `user1`:

```
user1@europa:~$ ls -C
documenti  esercizi  fotografie  programmi
user1@europa:~$ ls -C .
documenti  esercizi  fotografie  programmi
user1@europa:~$ cd esercizi/soluzioni
user1@europa:~/esercizi/soluzioni$ ls -C ../..
documenti  esercizi  fotografie  programmi
user1@europa:~/esercizi/soluzioni$ cd ..
user1@europa:~/esercizi$ ls -C ../..
user1 user2 user3
user1@europa:~/esercizi$
```

1.6.7 Creare e cancellare directory

I comandi per creare e cancellare le directory sono rispettivamente:

`mkdir d` crea una directory di nome `d`

`rmdir d` cancella la directory `d` purché essa sia vuota

Nell'esempio seguente vengono usati tali comandi per effettuare alcune operazioni dimostrative sulle directory di esempio:

```
user1@europa:~$ mkdir tmp
user1@europa:~$ ls -l
total 20
drwxr-xr-x 3 user1 user1 4096 2008-07-23 14:14 documenti
drwxr-xr-x 3 user1 user1 4096 2008-07-23 14:12 esercizi
drwxr-xr-x 2 user1 user1 4096 2008-07-23 14:12 fotografie
drwxr-xr-x 2 user1 user1 4096 2008-07-23 14:12 programmi
drwxr-xr-x 2 user1 user1 4096 2008-07-23 14:32 tmp
user1@europa:~$ mkdir tmp/d1/d2
mkdir: cannot create directory 'tmp/d1/d2': No such file or directory
user1@europa:~$ mkdir tmp/d1
user1@europa:~$ ls -l tmp/d1
total 0
user1@europa:~$ rmdir tmp
rmdir: failed to remove 'tmp/': Directory not empty
```

```
user1@europa:~$ rmdir tmp/d1
user1@europa:~$ ls -l tmp/d1
ls: cannot access tmp/d1: No such file or directory
user1@europa:~$
```

Si noti il fatto che sono stati impartiti alcuni comandi validi ma che, invece di portare all'esecuzione dell'operazione richiesta, hanno causato un messaggio di avvertimento. Con il comando

```
mkdir tmp/d1/d2
```

si è tentato di creare la directory `d2` come sottodirectory di `tmp/d1`; la directory `tmp` esiste, ma la sua sottodirectory `d1` no, e quindi è impossibile creare una directory `d2` contenuta in `d1` (quest'ultima non esiste!). Il comando

```
rmdir tmp
```

ha invece tentato di cancellare la directory `tmp` che però non è vuota, e quindi non si può cancellare. Infine, col comando

```
ls -l tmp/d1
```

si è tentato di elencare il contenuto della directory `tmp/d1`, che era stata precedentemente cancellata e quindi non esiste più.

Una nota importante per quanto riguarda la visualizzazione a terminale dei messaggi di errore: i messaggi di errore che raggiungono il terminale non vengono emessi sullo standard output ma su un altro canale, detto *standard error* (stderr). In questo modo quando l'output di un comando viene rediretto su file, eventuali messaggi di errore raggiungono comunque l'utente, venendo visualizzati sullo standard error che rimane associato al terminale.

1.6.8 Copiare file: cp

Per copia di un file si intende la creazione di un secondo file che contiene esattamente le stesse informazioni del file di partenza. Anche le directory, come i file, possono essere copiate.

La copia di file e directory si effettua utilizzando il comando `cp`. Il comando

```
cp f1 [f2 ...] dir
```

crea delle copie dei file `f1...` dentro la directory `dir`. Valgono le seguenti regole:

- `dir` deve essere una directory esistente
- se `f1` esiste già dentro `dir`, il file viene sovrascritto

Il comando

```
cp f1 f2
```

crea una copia del file `f1` di nome `f2` nella directory corrente. In questo caso

- `f2` deve essere un file o non esistere (altrimenti vedi caso precedente)
- se `f2` esiste già viene sovrascritto

```
user1@europa:~$ cd documenti/
user1@europa:~/documenti$ cp agenda contatti ../tmp
```

```
user1@europa:~/documenti$ ls -l ../tmp/
total 8
-rw-r--r-- 1 user1 user1 28 2008-07-23 14:51 agenda
-rw-r--r-- 1 user1 user1 19 2008-07-23 14:51 contatti
user1@europa:~/documenti$ echo domani partita a tennis >> agenda
user1@europa:~/documenti$ cp agenda ../tmp/
user1@europa:~/documenti$ ls -l ../tmp/
total 8
-rw-r--r-- 1 user1 user1 52 2008-07-23 14:52 agenda
-rw-r--r-- 1 user1 user1 19 2008-07-23 14:51 contatti
user1@europa:~/documenti$
```

Si noti che listando la seconda volta il contenuto della directory `tmp` si vede come il file `agenda` sia stato sovrascritto. Ha infatti una dimensione maggiore (52 bytes invece che 28), poiché contiene anche l'ultima stringa inserita.

È possibile copiare ricorsivamente una directory aggiungendo a `cp` l'opzione `-r`, cioè utilizzando il comando

```
cp -r f1 [f2 ...] dir
```

in questo caso, se `f1...` è una directory, essa viene copiata ricorsivamente, cioè insieme alle sue subdirectory e tutti i file contenuti.

```
user1@europa:~$ ls documenti/
affari/   agenda    contatti
user1@europa:~$ cp -r documenti tmp/
user1@europa:~$ ls -RC tmp/
tmp/:
documenti

tmp/documenti:
affari  agenda  contatti

tmp/documenti/affari:
user1@europa:~$
```

1.6.9 Spostare i file: `mv`

Lo spostamento di file e directory si effettua per mezzo del comando

```
mv f1 [f2 ...] dir
```

il quale sposta gli oggetti `f1...`, siano essi file o directory, dentro la directory `dir`. Valgono le seguenti regole:

- `dir` deve esistere come directory
- se `f1` esiste già dentro `dir` il file viene sovrascritto
- `f1 [f2 ...]` può essere una directory, la quale verrà copiata ricorsivamente dentro `dir`

Il comando `mv` viene anche utilizzato per rinominare file e directory. In particolare:

```
mv f1 f2
```

cambia il nome di `f1` in `f2`. In questo caso

- `f2` deve essere un file o non esistere (altrimenti vedi caso precedente)
- se `f2` esiste già viene sovrascritto

Un esempio di utilizzo di `mv`:

```
user1@europa:~$ ls tmp
user1@europa:~$ echo ciao > tmp/saluto
user1@europa:~$ ls tmp
saluto
user1@europa:~$ mv tmp/saluto .
user1@europa:~$ ls tmp
user1@europa:~$ ls
documenti  esercizi  fotografie  programmi  saluto  tmp
user1@europa:~$ ls tmp/saluto
ls: cannot access tmp/saluto: No such file or directory
user1@europa:~$ mv saluto tmp/ciao.file
user1@europa:~$ ls
documenti  esercizi  fotografie  programmi  tmp
user1@europa:~$ ls tmp
ciao.file
user1@europa:~$ cat tmp/ciao.file
ciao
user1@europa:~$
```

1.6.10 Cancellare file: `rm`

Per rimuovere uno o più file si utilizza il comando `rm`:

```
rm f1 [f2 ...]
```

il quale cancella i file `f1`, `f2`, ecc. Per esempio

```
user1@europa:~$ echo ciao > tmp/saluto
user1@europa:~$ ls tmp
saluto
user1@europa:~$ rm tmp/saluto
user1@europa:~$ ls tmp
user1@europa:~$
```

È possibile usare il comando `rm` anche per cancellare le directory. Il comando

```
rm -r dir
```

cancella ricorsivamente la directory `dir` insieme a tutte le sotto-directory e tutti i file contenuti.

L'argomento `-f` ("force") permette di imporre la cancellazione (o la sovrascrittura) senza una richiesta di conferma. Quindi il comando

```
rm -fr dir
```

che accorpa le opzioni `-f` e `-r` elimina la directory `dir` e tutto il suo contenuto senza chiedere alcuna conferma all'utente.

```
user1@europa:~$ ls -r documenti/
contatti  agenda  affari
user1@europa:~$ ls -R documenti/
documenti/:
affari  agenda  contatti

documenti/affari:
user1@europa:~$ rm -fr documenti/affari
user1@europa:~$ ls -R documenti/
documenti/:
agenda  contatti
```

Nell'esempio precedente, viene usato il comando `rm` per cancellare ricorsivamente la sottodirectory `affari` di `documenti`.

I comandi `rm`, `cp`, `mv`, poiché possono cancellare dei file o sovrascriverli (cancellando l'eventuale contenuto preesistente), permettono di specificare l'argomento `-i` ("interattivo"), nel qual caso viene chiesta conferma all'utente per ogni cancellazione effettuata.

NOTA In generale non c'è modo di recuperare i file una volta che sono stati cancellati: `rm *` e `rm -r dir` sono comandi molto pericolosi, a maggior ragione se vengono utilizzati congiuntamente.

Su molte macchine l'amministratore di sistema decide che `rm` è equivalente a `rm -i`, nel qual caso viene sempre chiesta conferma quando si cancellano i file.

1.7 LE WILDCARD

Di notevole utilità è l'uso delle cosiddette "wildcard", ovvero speciali caratteri che servono ad individuare più file in una sola invocazione di un comando. In particolare, sono disponibili le seguenti wildcard:

- `*` indica *tutti i caratteri*
- `?` indica *uno e un solo carattere*

Alcuni esempi serviranno a chiarirne le modalità di utilizzo.

Il comando

```
ls ag*
```

elenca tutti i file che iniziano per "ag", come ad esempio `agenda1`, `agenda2.txt`, `aggiornamento.dat`.
Il comando

```
rm ag*t
```

cancella tutti i file che iniziano con "ag" e finiscono con "t". Tra "ag" e "t" ci può essere un qualsiasi numero di caratteri (anche 0) qualsiasi. Ad esempio, vengono cancellati `agenda.txt`, `agt`, `ag_qualsiasi-testo.t`. Il file `agtx` non viene invece cancellato.
Il comando

```
rm -fr *
```

cancella tutti i file e le directory presenti nella directory corrente.

NOTA

ATTENZIONE: i due comandi

- `rm -fr documenti/*`
- `rm -fr documenti/`

sono leggermente diversi tra loro: il primo cancella ricorsivamente tutto *il contenuto* della directory `documenti`, ma NON CANCELLA la directory `documenti` stessa; il secondo comando cancella la directory `documenti` con tutto il suo contenuto.

Il comando

```
ls img?
```

elenca tutti i file che iniziano con i caratteri "img", seguiti da uno e un solo carattere. Per esempio, vengono elencati i file `img1`, `img2`, mentre non vengono elencati i file `img`, `img12`, `im_g`.

Un altro esempio, più vicino alla comune esperienza di programmazione in C, è il seguente:

```
ls programma.?
```

che elenca i file `programma.c`, `programma.h` e `programma.o`, i quali costituiscono i tipici file collegati al processo di realizzazione (scrittura del sorgente e compilazione) di un programma in C.

1.8 VISUALIZZAZIONE DEL MANUALE

È possibile utilizzare il comando `man` per visualizzare il manuale relativo a ogni comando disponibile in un sistema Unix, ovvero la relativa documentazione. Per esempio, il comando

```
man ls
```

visualizza la pagina del manuale per il comando `ls`.

Alcuni comandi sono interni all'interprete (*builtin* della shell), cioè non corrispondono a programmi fisicamente presenti sul disco. La gestione di questi comandi è programmata direttamente all'interno della shell. Esempi di comandi builtin sono `cd` e `pwd`. Per questo motivo, per esempio i comandi

```
man cd
man pwd
```

non danno risultati. Per conoscere i dettagli dei comandi interni basta invocare il comando

```
man sh
```

che visualizza la pagina di manuale del programma `sh`, il quale non è altro che la shell stessa.

1.9 ESECUZIONE DI COMANDI

I comandi disponibili in un sistema Unix non sono altro che dei programmi eseguibili fisicamente presenti sul disco, i quali vengono eseguiti quando vengono invocati da linea di comando. In tal senso, il compito principale della shell è quello di eseguire i programmi relativi ai comandi desiderati.

La maggior parte dei comandi più utilizzati risiede nelle directory di sistema

```
/bin
/usr/bin
```

quindi visualizzando il contenuto di tali directory con

```
ls /bin
ls /usr/bin
```

si possono scoprire molti comandi non descritti in questo documento.

In genere per l'esecuzione di un programma è necessario specificare esattamente il percorso (relativo o assoluto) del programma stesso. Per esempio, è possibile invocare il comando `rmdir` con la linea di comando:

```
/bin/rmdir directory-da-cancellare
```

Il fatto che i comandi illustrati finora non richiedano la specifica di tutto il percorso per essere invocati è dovuto al fatto che è possibile specificare alla shell alcune directory nella quale ricercare i comandi che vengono invocati senza specificare il percorso[4]. Non ci soffermeremo su come questa opportunità può essere sfruttata.

È però importante sottolineare la differenza tra due diverse linee di comando per l'invocazione programmi, che talvolta vengono confuse. Si considerino i due seguenti comandi:

```
$ ../rm nomefile

$ rm ../nomefile
```

Il primo comando invoca il programma `rm` presente nella directory `..` della directory corrente[5], passandogli come argomento la stringa `nomefile`. Il secondo comando invoca il programma `rm` presente nella directory di sistema e cancella (o tenta di cancellare, se il file non esiste...) il file di nome `nomefile` presente nella directory "`..`".

Ovviamente l'esempio proposto vale per il comando `rm`, ma il ragionamento si estende a qualsiasi altro comando o, più in generale, programma eseguibile.

[4]Per maggiori informazioni è necessario documentarsi su concetti quali le variabili di ambiente, e in particolare la variabile `PATH`.

[5]Noi stessi potremmo creare un programma che si chiama `rm` ben diverso da quello di sistema, e volerlo richiamare, anche se potrebbe aver poco senso farlo per evitare confusione con comandi standard.

Capitolo 2

DAL PROBLEMA AL PROGRAMMA, PASSANDO DALL'ALGORITMO

Uno degli scopi fondamentali dell'informatica è la risoluzione di problemi. Informalmente, per problema si intende un compito che si vuole far risolvere automaticamente a un calcolatore. I problemi di interesse sono solitamente parametrici, nel senso che dipendono da dati i cui valori non sono noti al momento in cui si vuole affrontare e risolvere il problema. Tali dati divengono quindi dei *parametri* da fornire alla procedura di risoluzione del problema al momento in cui questa viene eseguita.

Per risolvere un problema bisogna svolgere le seguenti attività:

- comprendere il problema
- definire un procedimento risolutivo (algoritmo) per il problema
- implementare l'algoritmo in un linguaggio di programmazione

È importante notare che la ricerca della procedura risolutiva di un problema non sempre è necessariamente il compito più complesso da svolgere. Si possono infatti avere problemi la cui soluzione è relativamente semplice, ma sono difficili da comprendere. La difficoltà nella comprensione può derivare da vari aspetti: il problema è specificato in termini poco chiari o poco formali; la specifica del problema può venire da ambienti completamente diversi da quelli a cui appartiene chi deve interpretare il problema stesso. Per esempio, un tecnico informatico potrebbe avere difficoltà a comprendere un problema formulato da un responsabile vendite di un'azienda, poiché il lessico e il modo di descrivere uno stesso problema è molto differente tra i due. In generale, la descrizione del problema non fornisce un metodo per calcolare il risultato.

Affinché un problema sia risolvibile, in generale è necessario che la sua definizione sia chiara e completa. Non tutti i problemi sono risolvibili attraverso l'uso del calcolatore. In particolare esistono classi di problemi per le quali la soluzione automatica non è proponibile. Ad esempio:

- il problema può presentare infinite soluzioni
- per alcuni problemi non è stato trovato un metodo risolutivo
- per alcuni problemi è stato dimostrato che non esiste un metodo risolutivo automatizzabile

Nel seguito ci si concentrerà su problemi che, ragionevolmente, ammettono un metodo risolutivo.

2.1 ESEMPI DI PROBLEMI

Alcuni esempi di problemi da risolvere sono:

- dati due numeri trovare il maggiore
- dati a e b, risolvere l'equazione $ax + b = 0$
- calcolare il massimo comun divisore fra due numeri dati
- dato un insieme di parole, metterle in ordine alfabetico
- dato un elenco di nomi e relativi numeri di telefono trovare il numero di telefono di una determinata persona
- dato l'archivio dell'anagrafe comunale, trovare tutti i nuclei familiari composti da più di 4 persone
- dato l'archivio dei dipendenti di un'azienda, calcolare lo stipendio di ogni dipendente dell'azienda

2.2 L'ALGORITMO

Per risolvere un problema bisogna definire, cioè identificare o progettare, un procedimento risolutivo, ossia un insieme di passi elementari, o istruzioni, che, eseguiti secondo un ordine prestabilito, permettono di arrivare ai risultati desiderati a partire dai dati del problema, cioè un *algoritmo*.

In campo informatico, l'algoritmo è definito come un metodo per risolvere un problema che sia adatto a essere implementato sotto forma di programma.

In generale, cioè se si prescinde dall'ambito informatico, un algoritmo è una qualsiasi sequenza di istruzioni che specifica in modo formale come realizzare un compito. In questo senso anche una ricetta di cucina, le istruzioni per far funzionare un elettrodomestico, le istruzioni per installare un programma sono esempi di algoritmi.

Un algoritmo deve quindi essere espresso in termini delle istruzioni di un esecutore automatico, molto spesso un calcolatore, cioè:

- ciascuna istruzione deve poter essere eseguita dall'esecutore in tempo finito
- l'intera sequenza di istruzioni deve poter essere eseguita in tempo finito, per ogni possibile insieme di ingresso che soddisfa la pre-condizione del problema

Comunemente, il termine algoritmo viene usato in campo matematico ed informatico. In questi contesti, una definizione più formale di algoritmo può essere la seguente:

sequenza logica di istruzioni elementari (univocamente interpretabili) che, eseguite in un ordine stabilito, permettono la soluzione di un problema in un numero finito di passi

In tal senso, esempi di algoritmi sono il calcolo del prodotto di matrici o l'ordinamento di un insieme di numeri.

2.3 PROPRIETÀ DI UN ALGORITMO

La definizione di algoritmo implica una serie di condizioni che devono essere valide per un algoritmo, ovvero:

1. *realizzabilità*: ogni operazione prevista dall'algoritmo deve essere eseguibile con le risorse a disposizione;
2. *non ambiguità*: ogni operazione deve essere univocamente interpretabile dall'esecutore, cioè deve essere descritta in modo preciso, ma sintetico per consentirne una interpretazione univoca. Questo implica che i risultati ottenuti mediante un algoritmo risolutivo di un problema non devono cambiare al variare dell'esecutore (macchina/persona) dell'algoritmo (carattere deterministico);
3. *finitezza*: il numero totale di istruzioni da eseguire, per ogni insieme di dati di ingresso, è finito e le operazioni da esse specificate devono essere eseguite un numero finito di volte.

Pertanto, l'algoritmo deve

- essere costituito da operazioni appartenenti ad un determinato insieme di operazioni fondamentali (sistema formale);
- essere applicabile a qualsiasi insieme dei dati di ingresso appartenenti al dominio di definizione dell'algoritmo.

Un algoritmo possiede inoltre due qualità fondamentali

1. la *correttezza*, ovvero l'algoritmo deve permettere effettivamente di risolvere il problema
2. l'*efficienza*, cioè l'esecuzione dell'algoritmo deve richiedere un uso limitato di risorse

Tipiche risorse che devono essere salvaguardate sono il tempo di esecuzione e la quantità di memoria utilizzata. Tra le qualità degli algoritmi si possono inoltre annoverare:

- la *leggibilità*, in quanto l'algoritmo essere facilmente comprensibile
- la *modificabilità*: l'algoritmo deve essere facilmente modificabile, a fronte di (piccole) modifiche nelle specifiche del problema risolto dall'algoritmo
- la *parametricità*, cioè l'algoritmo deve dipendere da dati i cui valori non sono noti al momento in cui si vuole affrontare e risolvere il problema
- la *riusabilità*

2.4 ESEMPI DI ALGORITMO

In questa sezione verrà mostrato il processo di risoluzione di alcuni semplici problemi.

2.4.1 Prodotto di matrici

Problema: calcolare il prodotto di due matrici A e B di dimensione rispettivamente $m \times n$ e $n \times p$.
Dati di ingresso: gli $m \cdot n$ valori che compongono A, che saranno indicati con a_{ij}; gli $n \cdot p$ valori che compongono B, che saranno indicati con b_{ij}.
Pre-condizione: deve essere verificato che il numero di colonne di A deve essere uguale al numero di righe di B.
Dati di uscita: una matrice C di dimensione $m \times p$.
Specifica dell'algoritmo:

- l'elemento c_{ij} della matrice C viene calcolato come

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \ldots + a_{in} b_{nj}$$

2.4.2 Calcolo del massimo comune divisore

Problema: calcolare il massimo comune divisore di due numeri interi.
Dati di ingresso: due numeri interi a e b, con $a > b$.
Pre-condizione: $a \neq 0$, $b \neq 0$.
Dati di uscita: il più alto numero intero c che divide sia a che b.
Specifica dell'algoritmo:

1. assegna ad r il resto della divisione a/b
2. se $r = 0$ allora l'MCD è pari a b
3. altrimenti assegna $a = b$ e $b = r$ e ripeti dal punto 1

L'algoritmo utilizzato è noto come *algoritmo di Euclide*.

2.5 LA PROGRAMMAZIONE

L'obiettivo della *programmazione* è quello di *implementare* un algoritmo, ovvero scrivere un *programma* che realizzi le operazioni specificate dall'algoritmo.

Nelle sezioni seguenti verranno approfonditi i concetti relativi all'attività della programmazione. Il *processore*, il componente più importante di un sistema di calcolo, si occupa proprio di eseguire le istruzioni che compongono il programma.

2.6 IL PROGRAMMA

Un programma è l'implementazione di un algoritmo in un linguaggio adatto a essere eseguito da un computer, o da un qualsiasi sistema automatico che interpreti le istruzioni di cui il programma è composto.

Il termine programma è spesso usato in modo intercambiabile con altri termini, come software o applicazione. Tale interpretazione può essere considerata errata in quanto, per esempio, una applicazione è talvolta composta da diversi programmi.

Le operazioni base che possono essere utilizzate per la realizzazione di un programma sono quattro:

1. *trasferimento di informazioni*: acquisizione dati, visualizzazione risultati intermedi, scrittura risultati finali

2. *esecuzione di calcoli*
3. assunzione di *decisioni*: scelta della successiva operazione da compiere sulla base di risultati intermedi
4. esecuzione di *iterazioni*: ripetizione di sequenze di operazioni

Per rappresentare (descrivere) un algoritmo non è possibile utilizzare il linguaggio naturale in quanto questo può presentare ambiguità che potrebbero causare interpretazioni false o errate. È necessario, pertanto, utilizzare linguaggi sintetici e standardizzati in modo da consentire all’esecutore una interpretazione univoca dei passi da svolgere.

2.7 CORRETTEZZA DI UN PROGRAMMA

La correttezza di un programma si valuta sotto due aspetti: la sintassi e la semantica.

La frase

l’area di un rettangolo e base per altezza

è corretta dal punto di vista sintattico, un quanto è composta dalle due frasi di senso compiuto *l’area di un rettangolo* e *base per altezza*, unite dalla congiunzione *e*. Una frase del genere è sintatticamente corretta poiché le regole della lingua italiana non ne vietano la costruzione. Non è però corretta dal punto di vista semantico: sostanzialmente non significa nulla.

Una frase corretta sia sintatticamente che semanticamente si ottiene con un cambiamento minimo:

l’area di un rettangolo è base per altezza

2.8 ESECUZIONE DEL PROGRAMMA

L’*esecuzione* di un programma è la fase nella quale le istruzioni che compongono il programma vengono decodificate e le corrispondenti operazioni vengono eseguite.

Le tipiche operazioni che hanno luogo quando un programma viene eseguito sono le seguenti:

- caricamento in memoria delle istruzioni che compongono il programma, tipicamente a partire da una periferica di memoria di massa, come un disco rigido
- identificazione del “punto d’ingresso” del programma (inizializzazione del Program Counter)
- caricamento nei registri (fetch), decodifica ed esecuzione sequenziale delle istruzioni

2.9 ERRORI E DEBUG

Durante la scrittura del codice che costituisce un programma, capita inevitabilmente di commettere errori, sia di tipo sintattico, che semantico, che logico. L’errore di programmazione viene universalmente chiamato *bug*.

Il *debugging* è quindi la fase dello sviluppo nella quale si ricercano e correggono gli errori.

Gli errori di sintassi sono relativamente semplici da trovare, in quanto infrangono delle regole ben definite per la scrittura del codice. Sono segnalati in modo automatico dagli strumenti usati per lo sviluppo di un programma. Alcuni esempi sono:

- scrivere *wile* invece di *while*

- dimenticare di chiudere una parentesi precedentemente aperta

Gli errori semantici vengono tipicamente segnalati in modo automatico, una volta che il programma è sintatticamente corretto. Per esempio:

- richiamare una funzione che non esiste
- confrontare un numero intero con una stringa
- assegnare un valore ad una costante (3 = x)
- assegnare un valore ad una espressione (x + y = 3)

Gli errori logici sono più difficili da identificare, in quanto sono collegati, appunto, alla logica di funzionamento del programma. Per questo motivo è difficile rilevarli per mezzo di procedure automatiche. Inoltre si manifestano tipicamente in fase di esecuzione del programma, cosa che complica ulteriormente il debugging. In altri termini, si potrebbe pensare agli errori logici come ad un'implementazione sintatticamente e semanticamente corretta di un algoritmo sbagliato, cioè si tratta di un algoritmo che non realizza le operazioni attese. Esempi di errori logici sono:

- effettuare un ciclo per un numero di volte errato
- combinare in modo errato più test nelle istruzioni condizionali

2.10 TESTING

Il *testing* è la fase nella quale un programma realizzato ed eseguibile, che quindi è corretto dal punto di vista sintattico e semantico, viene collaudato per verificarne la correttezza logica, ovvero si verifica che l'algoritmo implementato svolga correttamente le operazioni previste.

Il testing si realizza generalmente fornendo in ingresso al programma opportuni valori di input per verificare che l'output corrispondente sia corretto.

2.11 MANUTENZIONE

Un aspetto talvolta trascurato della programmazione è relativo alla *manutenzione*. Spesso il programmatore non deve sviluppare *ex novo* un programma, ma si trova a dover *modificare* un programma esistente.

Il problema nasce dal fatto che spesso il programma è stato scritto da altri programmatori, oppure è stato scritto dallo stesso programmatore, ma è passato un periodo di tempo sufficiente da far dimenticare i dettagli dell'implementazione.

La manutenzione di un programma è notevolmente facilitata se si utilizza uno stile di programmazione chiaro (vedi Sezione 15) e si commenta opportunamente il codice, in corrispondenza dei punti che possono risultare particolarmente difficili da interpretare.

2.12 LINGUAGGI DI PROGRAMMAZIONE

Un programma viene realizzato scrivendo il cosiddetto *codice sorgente* utilizzando un linguaggio di programmazione (Sezione 2.6), ovvero un linguaggio con regole sintattiche ben definite, che permettono di interpretare univocamente le specifiche del programmatore. Il codice sorgente viene memorizzato in file di testo, detti *file sorgente* o, più sinteticamente, *sorgenti*.

Esistono molti diversi linguaggi di programmazione, ciascuno dei quali ha caratteristiche specifiche che lo rendono adatto a compiti specifici. Quindi per ciascuna tipologia di applicazione (es. web, accesso a database, sistemi operativi, sistemi embedded, scripting, ambienti di simulazione, interfacce grafiche, calcoli matematici, applicazioni di controllo, ecc.) esiste almeno un linguaggio appositamente sviluppato per effettuare le particolari operazioni specifiche dell'applicazione in modo più semplice e diretto di quanto si potrebbe fare con altri linguaggi. Perciò, in generale, un solo linguaggio è poco flessibile per poter essere utilizzato in contesti molto diversi tra loro.

Per questo motivo i linguaggi di programmazione vengono classificati in vari modi. Le principali categorie di linguaggi di programmazione sono le seguenti:

- linguaggi interpretati vs compilati
- linguaggi di basso livello vs alto livello
- linguaggi procedurali
- linguaggi funzionali
- linguaggi dichiarativi
- linguaggi ad oggetti
- linguaggi di scripting

2.13 EVOLUZIONE DEL LINGUAGGIO C

Una breve storia dell'evoluzione del C:

- Martin Richards sviluppa il BCPL, pensato per scrivere sistemi operativi e software di sistema
- alcune caratteristiche del BCPL sono ereditate dal linguaggio B, anch'esso sviluppato con lo stesso scopo da Ken Thompson nel 1970 per il primo sistema UNIX
- 1972: Dennis Ritchie progettava e realizzava, presso i Bell Laboratories, la prima versione del linguaggio C
- gli stessi Ritchie e Thompson riscrissero in C il codice di UNIX
- inizialmente UNIX viene utilizzato solo nei Laboratori Bell (come ambiente di sviluppo del software), quindi nell'università californiana di Berkeley (UCB). In questi due ambienti UNIX si sviluppa fino a diventare uno dei sistemi più completi sul mercato. Il C si sviluppa e si diffonde parallelamente a UNIX
- 1983: l'Istituto Nazionale Americano per gli Standard (ANSI), costituisce un comitato per definire in modo completo il linguaggio e l'insieme minimo di funzioni di libreria che un compilatore deve implementare
- 1989: è approvato lo standard ANSI o ANSI C
- 1995: adottato l'Emendamento 1 al C Standard che, fra le altre cose, ha aggiunto nuove funzioni alla libreria standard del linguaggio
- A partire dal C89 con l'Emendamento 1, e unendovi l'uso delle classi di Simula, Bjarne Stroustrup inizia lo sviluppo del C++
- 1999: promulgazione dello standard ISO C99 (codice ISO 9899)

Capitolo 3

REALIZZAZIONE DI UN PROGRAMMA

La realizzazione di un programma ha come obiettivo la generazione del cosiddetto *file eseguibile*, ovvero di un file che contiene le istruzioni codificate in linguaggio macchina.

Il file eseguibile è così chiamato in quanto è pronto per l'esecuzione da parte del processore. Non appena il programma viene caricato nella memoria del computer dal sistema operativo, il processore può iniziare a decodificarne le istruzioni e, quindi, ad eseguirle.

In Figura 3.1 è riportato il flusso logico delle operazioni necessarie per passare dal codice sorgente del programma al file eseguibile. I singoli passi rappresentati in figura saranno descritti in dettagli in questo capitolo.

Nonostante la descrizione del processo di realizzazione di un programma presentato nel Capitolo 2 sia piuttosto generale, e può essere quindi ritenuta valida per qualsiasi linguaggio di programmazione, lo schema di Figura 3.1, che mostra le fasi necessarie a passare da un file sorgente ad un programma eseguibile, non è invece valido per tutti i linguaggi di programmazione.

Esistono infatti linguaggi che non necessitano del processo di compilazione. Infatti, in alcuni linguaggi il codice sorgente viene interpretato da un programma chiamato *interprete*, che traduce il sorgente in codice macchina una istruzione dopo l'altra, ed esegue le instruzioni durante il processo di traduzione. Il più noto esempio di linguaggio interpretati è il linguaggio BASIC.

Altri linguaggi utilizzano una tecnica mista. Il linguaggio Java, ad esempio, utilizza il processo di compilazione per generare un file che non è codificato in linguaggio macchina, ma in un codice intermedio chiamato bytecode. L'esecuzione vera e propria del programma viene affidata ad un interprete che traduce il bytecode e lo trasforma in codice macchina. Questo approccio permette di rendere i programmi Java particolarmente *portabili* tra sistemi operativi e architetture diverse, in quanto basta realizzare un interprete specifico per ciascuna piattaforma per essere in grado di eseguire

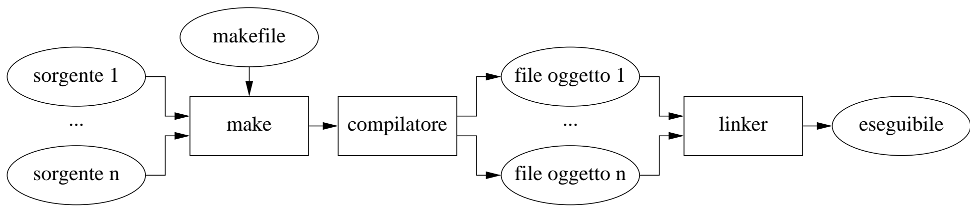


Figura 3.1 Schema descrittivo della realizzazione di un programma, che mostra le fasi necessarie a passare dal codice sorgente al programma eseguibile tipiche del linguaggio C.

un programma Java. In questo caso, infatti, il codice intermedio (bytecode), è ancora indipendente dalla piattaforma, e quindi maggiormente portabile.

3.1 IL FILE SORGENTE

La scrittura di un programma prevede la stesura del cosiddetto *codice sorgente*, ovvero delle istruzioni scritte nel linguaggio di programmazione.

Generalmente il file sorgente è memorizzato sotto forma di file di testo, e non sono quindi necessari editor particolari per la stesura del codice.

3.2 IL COMPILATORE

Il *compilatore* è un programma che accetta in input un file sorgente e restituisce in uscita l'equivalente programma in codice macchina, generando il cosiddetto *file oggetto*.

Il compilatore prende il codice C, lo fa elaborare tipograficamente dal preprocessore e lo traduce in codice assembler, lo passa quindi all'assemblatore per ottenere dei file *oggetto*, contenenti codice macchina.

Il compilatore legge il codice sorgente una volta sola, quindi in certi casi occorre dichiarare una variabile o una funzione, oltre a definirla. Per esempio, se una funzione ne chiama un'altra definita più avanti nello stesso file sorgente, occorre dichiarare in anticipo tale funzione. Le dichiarazioni delle funzioni di libreria vengono raccolte in file *header* (intestazioni) che vengono inclusi all'inizio di ogni sorgente C.

3.3 IL LINKER E I FILE OGGETTO

Un file oggetto contiene codice e dati unitamente ad elenchi di *simboli* non definiti. Un simbolo, a questo livello, è solo un nome a cui deve essere associato un indirizzo di memoria. Il nome è associato a variabili, funzioni, ecc. La risoluzione finale dei simboli non definiti viene effettuata da un programma chiamato *linker* il cui file eseguibile è *ld*.

Un ruolo importante è svolto dal linker quando un programma è costituito da più file sorgente. In tal caso, il compilatore genera un file oggetto per ciascun file sorgente, ed è quindi compito del linker la generazione del programma complessivo composto dai diversi file oggetto. In questa fase il linker si preoccupa di verificare che tutti i simboli utilizzati nei diversi file siano stati correttamente definiti e/o inclusi.

Alcuni errori di compilazione, quindi, vengono riportati dal linker e non dal compilatore vero e proprio. Per esempio, quando il sorgente C contiene un nome di funzione digitato erroneamente

il linker riporta un errore di *undefined symbol*. A seconda di quante informazioni simboliche sono presenti nei file oggetto, il messaggio di errore può riferirsi ad una riga specifica di uno specifico file sorgente o mancare di riferimenti precisi al codice sorgente. Per "informazioni simboliche" si intendono infatti delle informazioni che, sulla base delle opzioni con cui viene invocato il comando di compilazione, possono o meno essere presenti nel file oggetto. Queste informazioni non sono necessarie per l'esecuzione del programma, ma servono per il debugging del programma stesso qualora si verifichino degli errori in fase di linking. Dal momento che il linker lavora sui file oggetto, e quindi non ha conoscenza del codice sorgente dal quale i file oggetto sono generati, non può produrre messaggi di errore che contengano per esempio informazioni sulla linea del file sorgente alla quale si è verificato l'errore, a meno che nel file oggetto non siano presenti opportune "informazioni simboliche" addizionali che mettano in relazione il codice oggetto con il relativo codice sorgente.

3.4 IL LOADER

Il *loader* è un componente del sistema operativo che si occupa del caricamento in memoria dei programmi eseguibili. Il loader prepara l'eseguibile per l'esecuzione e successivamente ne inizia l'esecuzione stessa. Il codice operativo del loader viene generalmente caricato in memoria all'avvio del sistema operativo, in attesa di essere utilizzato per l'esecuzione di un programma.

Tutti i sistemi operativi che supportano il caricamento di programmi eseguibili hanno un loader. Ci sono però sistemi, come per esempio i sistemi embedded, nei quali il loader non è presente. In tali sistemi viene eseguito sempre un solo programma, e non è quindi necessario un meccanismo per il caricamento di programmi diversi.

In ambiente Unix il loader svolge le seguenti funzioni, ogni qualvolta un programma deve essere eseguito:

1. effettua la validazione del programma (controllo dei permessi, requisiti di memoria, ecc.)
2. copia il programma dal disco alla memoria centrale
3. copia il contenuto della linea di comando sullo stack, in modo che il programma possa eventualmente accedervi
4. inizializza i registri
5. carica nel Program Counter l'indirizzo della prima istruzione da eseguire

3.5 STRUTTURAZIONE DEL CODICE

Un aspetto molto importante e talvolta trascurato nella programmazione è quello della strutturazione del codice sorgente. Per strutturazione del codice si possono intendere due aspetti. Uno più strettamente legato alla chiarezza "estetica" di presentazione del programma, e verrà descritto più in dettaglio alla Sezione 15; l'altro legato alla corretta suddivisione del codice in blocchi logici.

Il C appartiene alla categoria dei linguaggi cosiddetti "strutturati" in quanto permette di separare logicamente il flusso di un programma, implementando blocchi funzionali (le *funzioni*, per l'appunto), che realizzano operazioni semplici, e che vengono richiamate una o più volte all'interno del programma.

Uno degli aspetti più critici per chi si avvicina alla programmazione, in C ma non solo, è quella di organizzare logicamente il programma dividendolo opportunamente in funzioni.

In generale, si può implementare sotto forma di funzione qualsiasi operazione che debba essere *richiamata più di una volta* all'interno del programma. Talvolta questa regola non viene seguita per motivi molto particolari, ma è bene osservarla sempre quando possibile.

Alcune regole empiriche sono state elaborate per aiutare a capire se si sta dando la giusta impostazione al codice del programma. Per esempio è bene scrivere funzioni relativamente "corte", la cui lunghezza massima sia orientativamente quella di una schermata. Funzioni lunghe sono spesso suddivisibili in diverse funzioni più semplici.

Nel resto delle dispense verrà presentato più di un esempio per rendere chiaro il significato di suddivisione funzionale di un programma.

Capitolo 4

CONCETTI DI BASE

Tradizionalmente il C viene considerato un linguaggio di alto livello, anche se oggigiorno esistono linguaggi ad un livello ancora più elevato, come ad esempio il linguaggio Java. Anche se queste distinzioni hanno carattere prevalentemente storico, il C si può collocare ad un livello intermedio tra l'assembler e i linguaggi visuali più evoluti.

In generale, il linguaggio C è molto vicino alla macchina. È stato pensato come sostituto dell'assembler per aumentare la portabilità dei programmi; la traduzione in codice macchina risulta molto diretta e le tecniche di ottimizzazione del codice sono ben sviluppate. A causa della vicinanza al codice macchina, è il linguaggio più usato per la scrittura dei nuclei di sistema operativo e altre operazioni di basso livello, come driver per il controllo di periferiche.

Gli *oggetti* trattati dal linguaggio sono tutti oggetti semplici. In pratica sono tutti numeri interi, preferenzialmente della dimensione dei registri del processore in uso. Non esiste il concetto di *oggetto*, di *classe* e di *istanza* tipiche di linguaggi più evoluti come Java o C++. È però possibile, come risulterà chiaro nel seguito, adottare uno stile di programmazione orientata agli oggetti nella stesura dei propri programmi – ed è sempre raccomandabile farlo.

Il C è un linguaggio procedurale, ogni programma è perciò espresso come una sequenza di procedure che vengono dette funzioni. Ogni funzione è visibile globalmente in tutto l'applicativo, riceve un certo numero di argomenti e ritorna un solo valore oppure nessuno.

Il C è un linguaggio *dichiarativo* e *tipizzato*, nel senso che il *tipo* di ogni variabile (es. numero intero, virgola mobile, ecc.) deve essere esplicitamente *dichiarato* dal programmatore.

4.1 IL PRIMO PROGRAMMA IN C

Un esempio molto semplice di programma in C è il seguente[6].

Il programma è contenuto nel file hello.c.

NOTA — Per indicare che si tratta di un file sorgente in linguaggio C l'estensione del file deve essere .c[7].

```
1  /*
2   * programma che stampa sul video la frase
3   * Salve, mondo!
4   */
5  #include <stdio.h>
6
7  int main()
8  {
9    printf("Salve, Mondo!\n");
10   return 0;
11 }
```

NOTA — I numeri presenti a sinistra di ciascuna linea di codice non fanno parte del file sorgente, ma sono utilizzati per indicare in modo semplice una linea specifica.

Il programma di esempio presenta un insieme di caratteristiche del linguaggio che ora verranno soltanto elencate, e che verranno successivamente analizzate in dettaglio. Tali caratteristiche si annoverano:

- la presenza di un commento, alle linee [1...4], che in questo caso illustra l'obiettivo del programma;
- l'istruzione #include (linea 5) per includere un file di intestazione;
- la definizione della funzione main, che inizia alla linea 7 e termina alla linea 10;
- alla linea 9 compare l'istruzione printf per la stampa a video di una stringa di testo[8], quest'ultima delimitata dai doppi apici; da notare la presenza del carattere '\n', che corrisponde all'andata a capo (tutti i caratteri di una stringa che sono preceduti dal carattere \ – il *backslash* – hanno un significato particolare);
- infine, l'istruzione return alla linea 10 per ritornare il valore di uscita di una funzione (0 in questo caso).

L'esecuzione inizia dalla funzione main, che deve essere sempre presente in un programma C. La funzione main potrebbe ricevere alcuni argomenti, che per ora ignoriamo, e ritorna un numero intero. Il fatto che venga ritornato un valore di tipo intero è indicato dalla parola chiave int posta prima di main. Inoltre, il "ritornare un valore" significa che quando la funzione termina ovvero

[6]Il programma che stampa la stringa "Hello, world!" è tradizionalmente riportato come primo programma di esempio quando si presenta un linguaggio di programmazione.
[8]Una stringa di testo è sequenza di caratteri.

viene incontrata una istruzione `return` oppure si raggiunge l'ultima istruzione della funzione, viene restituito un opportuno valore intero specificato nell'istruzione `return` stessa. Quando `main` termina il programma termina a sua volta. Per convenzione, in genere se `main` ritorna zero vuol dire che il programma ha avuto successo nell'effettuare le operazioni previste, se ritorna non-zero vuol dire che c'è stato un problema nell'esecuzione; il numero specifico può indicare il tipo di problema riscontrato, se chi ha eseguito questo programma sa come differenziarli. Nell'esempio è presente una sola istruzione di `return` che ritorna zero: in questo semplice programma di esempio non è previsto che vengano commessi errori!

4.2 COMPILAZIONE DEL PROGRAMMA

Supponiamo di memorizzare il programma realizzato in un file chiamato `hello.c`. È sempre bene denominare un file sorgente C dandogli un nome significativo, che rispecchi il contenuto del programma che vi è implementato (ma questo è una prassi generale da utilizzare per la denominazione di qualsiasi file!).

Detto ciò, il comando per la compilazione del programma è il seguente:

```
$ gcc -Wall -o hello hello.c
```

Nell'istruzione precedente si possono distinguere le seguenti parti:

- l'invocazione del compilatore `gcc` (potrebbe anche essere `cc`, che in molte macchine non è altro che una chiamata a `gcc`)
- l'opzione `-Wall` che indica al compilatore di produrre tutti (`all`) i messaggi di avvertimento possibili, i cosiddetti (W)arnings
- l'opzione `-o hello`, che istruisce il compilatore a generare un file eseguibile denominato `hello` (`o` sta per *output*)
- l'indicazione del file sorgente da compilare `hello.c`

Se non fosse specificata l'opzione `-o`, il compilatore produrrebbe un file eseguibile denominato con il nome standard `a.out`. Se non fosse specificata l'opzione `-Wall`, il compilatore genererebbe solo i messaggi di avvertimento per i casi più "gravi", mentre non visualizzerebbe avvertimenti in grado di produrre spesso effetti comunque indesiderati.

I messaggi del compilatore possono essere infatti di due tipi:

1. segnalazioni di *errore*
2. segnalazione di *avvertimento* (warnings).

Nel caso siano presenti degli errori, tipicamente dovuti a errori di sintassi, la compilazione termina prematuramente e non viene quindi generato il file eseguibile. Nel caso di warnings, invece, la compilazione termina correttamente, producendo il file eseguibile. Un messaggio di avvertimento segnala al programmatore che ci potrebbe essere un errore di tipo semantico all'interno del programma (quelli di sintassi, si è detto, generano un errore). In alcuni casi il programma funziona correttamente anche in presenza di warnings, per esempio quando la segnalazione riguarda variabili dichiarate ma non utilizzate. In altri casi i problemi sono più insidiosi. Perciò, come regola generale, è bene realizzare programmi che generino quanti meno messaggi di avvertimento possibile.

Un esempio di esecuzione del programma di esempio è la seguente:

```
$ ./hello
Salve, Mondo!
```

4.2.1 Errori ed interpretazione dei messaggi del compilatore

Capita sovente nello sviluppo di un programma di commettere piccoli e grandi errori che vanno corretti per ottenere un programma perfettamente funzionante. Alcuni di questi errori sono segnalati dal compilatore, come precedentemente accennato. È quindi fondamentale saper bene interpretare il significato delle segnalazioni del compilatore in modo da poter risalire velocemente ed efficacemente alla sorgente del problema.

I messaggi di errore, e le tipologie di errori segnalati cambiano a seconda del compilatore utilizzato e, a parità di compilatore, in base alle diverse versioni dello stesso. Utilizzando il `gcc`, per conoscere la versione del compilatore in uso, è sufficiente il comando

```
$ gcc -v
gcc version 4.3.3
```

In genere sono stampate a video tutta una serie di altre informazioni sul compilatore, ma ci limitiamo ad indicare la versione del compilatore in uso. Nel seguito saranno infatti riportati e commentati alcuni esempi di errori generati dal compilatore che sono stati ottenuti con il compilatore `gcc` nella versione sopra riportata.

Considerando l'esempio del programma `hello.c`, si supponga di aver digitato l'istruzione alla linea 10 come segue

```
  return 0
```

nella quale si nota la mancanza del punto-e-virgola.

Se si compila il relativo programma, si ottiene il seguente risultato:

```
# gcc -Wall hello.c
hello.c: In function 'main':
hello.c:11: error: expected ';' before '}' token
```

Il compilatore segnala un errore, comunicando che nel file `hello.c`, alla linea 11 nella funzione `main` si attendeva un punto-e-virgola prima della parentesi graffa chiusa. In effetti tra lo zero che compare dopo la `return` e la parentesi graffa chiusa manca il punto-e-virgola. Perché il compilatore ci rimanda alla riga 11 se sappiamo che il punto-e-virgola è stato omesso nella riga precedente? Il compilatore legge il codice sorgente dall'inizio alla fine per convertirlo nel programma eseguibile, e si accorge dell'assenza del punto-e-virgola soltanto quando, leggendo la riga 11, incontra la parentesi graffa chiusa.

Un esempio un po' meno critico è dato dal fatto di omettere l'istruzione `return`, avendo cioè il seguente codice:

```
1  /*
2   * programma che stampa sul video la frase
3   * Salve, mondo!
4   */
5  #include <stdio.h>
6
7  int main()
8  {
9    printf("Salve, Mondo!\n");
10 }
```

La sua compilazione non genera un errore, ma un messaggio di avvertimento:

```
gcc -Wall hello.c
```

```
hello.c: In function 'main':
hello.c:10: warning: control reaches end of non-void function
```

Il compilatore ci informa che alla linea 10 della funzione `main` nel file `hello.c` si giunge alla fine della funzione senza aver incontrato una istruzione `return` che specifichi il valore di ritorno della funzione. Dal momento che la funzione `main` è stata definita per ritornare un valore intero, questo potrebbe causare dei problemi. Ma trattandosi di un warning, il programma eseguibile viene comunque generato correttamente.

Si supponga ora di aver digitato, invece della `return`, la seguente istruzione:

```
  ret 0;
```

La compilazione produce il seguente risultato:

```
gcc -Wall hello.c
hello.c: In function 'main':
hello.c:10: error: 'ret' undeclared (first use in this function)
hello.c:10: error: (Each undeclared identifier is reported only once
hello.c:10: error: for each function it appears in.)
hello.c:10: error: expected ';' before numeric constant
hello.c:11: warning: control reaches end of non-void function
```

Il compilatore non riconosce il termine `ret` alla linea 10 come un elemento valido. Non essendo una parola chiave (Sezione 4.6), assume che si tratti di una variabile (Sezione 4.7), la quale non è stata dichiarata prima di essere utilizzata.

Infine, un esempio molto istruttivo è quello nel quale l'istruzione 9 viene erroneamente digitata come

```
  printf("Salve, Mondo!\n);
```

cioè omettendo il doppio apice che delimita la stringa. Il risultato della compilazione è il seguente:

```
$ gcc -Wall hello.c
hello.c:9:10: warning: missing terminating " character
hello.c: In function 'main':
hello.c:9: error: missing terminating " character
hello.c:10: error: expected expression before 'return'
hello.c:11: error: expected ';' before '}' token
hello.c:11: warning: control reaches end of non-void function
```

È importante notare che questo errore alla linea 9 viene correttamente segnalato dal compilatore. Ma questo unico errore causa la generazione di altri due messaggi di errore (ed un messaggio di avvertimento, ma lo trascuriamo) alle linee successive (10 e 11). Ovviamente, sistemando l'errore di digitazione e aggiungendo correttamente il doppio apice al termine della stringa alla linea 9 risolve in un solo colpo tutti gli errori generati dal compilatore. In un programma più lungo e complesso, questo errore avrebbe potuto portare alla generazione di decine di messaggi di errore.

Questo banale esempio mette in evidenza che, in presenza di più messaggi di errore generati dal compilatore, si deve sempre cominciare ad analizzare il primo errore segnalato, in quanto il fatto di sistemare un erroe può rimuovere più di un messaggio di errore relativo a successive linee di codice nel sorgente. Chiaramente un programmatore esperto, che si assume sappia ciò che sta facendo, può permettersi di sistemare errori anche "in ordine sparso", se questo può risultargli in qualche modo conveniente. Ma per evitare problemi, è buona prassi correggere gli errori partendo dalla linea corrispondente al primo messaggio di errore del compilatore ed iterando il procedimento. Ovvero, corretto un errore, si procede alla successiva compilazione e si correggono gli eventuali altri errori.

4.3 I COMMENTI

I commenti sono porzioni di testo inserito in un programma che non vengono considerate dal compilatore al momento di generare il file oggetto. Vengono inseriti normalmente per permettere una più facile lettura e comprensione del codice.

I commenti sono di due tipi:

1. delimitati da /* e */
2. si estendono da // a fine riga.

La seconda forma viene dal C++, e molti programmatori C non la apprezzano.

Nell'esempio seguente sono utilizzate entrambe le tipologie di commento.

```
/*
 * questo e' un commento
 */

printf("Salve, Mondo!\n");  // anche questo e' un commento
printf("Salve, Mondo!\n");  /* e questo pure */
```

È sempre buona norma commentare adeguatamente i propri programmi spiegando come e perché il programma fa una certa cosa.

4.4 NOTA STILISTICA

Insieme ai commenti, uno degli aspetti che contribuisce di più alla realizzazione di programmi comprensibili e quindi più semplici da esaminare e manutenere anche a distanza di tempo, è lo stile col quale vengono organizzate le istruzioni del programma dal punto di vista "estetico".

Si noti che il programma seguente

```
#include <stdio.h>
int main(){printf("Salve, Mondo!\n");return 0;}
```

è, dal punto di vista funzionale, esattamente identico al programma di esempio iniziale. Infatti è una pura convenzione stilistica il fatto di scrivere le istruzioni su righe diverse, facendo opportunamente rientrare, ovvero *indentando*, le righe stesse. Si noti infatti nel programma `hello.c` originale che le istruzioni all'interno della funzione `main` sono fatte rientrare con l'aggiunta di due spazi. In questo modo risulta chiaro a colpo d'occhio che esse fanno parte della funzione stessa.

Lo spazio e l'andata a capo sono considerati dal compilatore allo stesso modo come separatori degli elementi che compongono il programma. Sia l'andata a capo che l'indentazione servono quindi per identificare a colpo d'occhio i blocchi di un programma che non sono altrimenti distinguibili. Tali blocchi possono essere ad esempio le istruzioni eseguite in un ciclo o all'interno di un blocco condizionale, che sono pertanto logicamente correlate.

4.5 GLI IDENTIFICATORI

Un *identificatore* è un nome che viene assegnato a funzioni, variabili, ed oggetti in genere definiti (o ridefinibili) dal programmatore stesso. Le regole che si applicano agli identificatori sono:

1. possono essere composti da lettere, cifre (caratteri alfanumerici) e sottolineature "_" (*underscore*)

2. non possono essere delle parole chiave (vedi Sezione 4.6)

3. il primo carattere non può essere numerico

4. il compilatore C distingue caratteri maiuscoli e minuscoli

Sono identificatori validi (e tutti diversi tra loro!):

```
Prova_1, prova_1,
totale_percentuale, _tot
```

Non sono invece validi:

```
1_prova, totale_%, un$bucato
```

La violazione delle regole applicabili agli identificatori costituiscono un errore di sintassi.

L'utilizzo di identificatori formati da più parole è spesso di aiuto per la comprensione del significato e del possibile utilizzo della corrispondente variabile. Per esempio, l'identificatore

```
OrdinaVettoreDiNomi
```

sarà verosimilmente associato ad una funzione che effettua l'ordinamento su un vettore di stringhe che rappresentano dei nomi.

Esistono due modi principalmente utilizzati per separare le parole in un identificatore che ne contenga più di una:

1. scrivere tutte le parole attaccate tra loro, utilizzando solo lettere minuscole, tranne la prima lettera di ciascuna parola che è maiuscola

2. scrivere le parole tutte minuscole e separate da sottolineature

Per esempio, l'identificatore precedente si potrebbe anche scrivere

```
ordina_vettore_di_nomi
```

mantenendo un eguale livello di "espressività" del codice.

NOTA L'uso di lettere maiuscole, per esempio in nomi come `OrdinaVettoreDiNomi`, rallenta la scrittura su tastiera e la lettura ad alta voce dei programmi. Anche se si tratta di un fatto più che altro stilistico, può essere preferibile l'uso dell'underscore.

4.6 LE PAROLE CHIAVE

Il linguaggio C riserva una serie di termini, detti *parole chiave*, per la definizione dei costrutti propri del linguaggio.

L'elenco completo delle parole chiave è il seguente:

`auto`, `break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `int`, `long`, `register`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, `while`.

L'aspetto più importante relativo alle parole chiave è che queste non possono essere utilizzate come identificatori, quindi non possono per esempio essere usate come nomi di variabili o funzioni.

Per questo motivo le parole chiave sono anche dette *parole riservate*, in quanto il loro utilizzo è riservato per la definizione del linguaggio stesso.

Il motivo per cui le parole chiave non possono essere utilizzate come identificatori è dato dal fatto che, se ciò fosse possibile, diventerebbe complicato per il compilatore, se non impossibile, ricostruire la struttura di un programma.

Alcune parole chiave sono utilizzate per realizzare i costrutti di controllo, descritti al Capitolo 5, in particolare:

`break`, `case`, `continue`, `default`, `do`, `else`, `for`, `goto`, `if`, `return`, `switch`, `while`.

Altre parole chiave sono invece inerenti ai tipi di dati forniti dal C, alla dichiarazione e all'uso di variabili e funzioni. Queste sono:

`auto`, `char`, `const`, `double`, `enum`, `extern`, `float`, `int`, `long`, `register`, `short`, `signed`, `sizeof`, `static`, `struct`, `typedef`, `union`, `unsigned`, `void`, `volatile`.

Gran parte parole chiave sono descritte nel Capitolo 6.

4.7 LE VARIABILI

Una *variabile* è tecnicamente una porzione di memoria che contiene dei dati *che possono essere modificati* nel corso dell'esecuzione del programma.

Ogni variabile deve essere dichiarata, ovvero *associata ad un identificatore*, e ad un tipo di dati.

Il programma è contenuto nel file `somma_semplice.c`.

Il seguente programma utilizza delle variabili per memorizzare valori necessari per calcolare la somma di due valori:

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int a, b, c;
6
7    a = 10;
8    b = 12;
9    c = a + b;
10   printf("La loro somma e' %d\n", c);
11
12   return 0;
13 }
```

Il programma dichiara, alla linea 5, le 3 variabili `a`, `b` e `c` di tipo `int` (intero), assegna un valore ad `a` e `b` (linee 7 e 8), calcola la loro somma, assegnandola alla variabile `c` (linea 9) e, alla linea 10, stampa a video il valore della somma.

L'operatore di assegnamento = (uguale) viene utilizzato per memorizzare nella variabile presente sulla sua sinistra il valore calcolato dall'espressione alla sua destra. Per esempio, l'istruzione

```
a = 12
```

assegna il valore costante 12 alla variabile `a`. Dal punto di vista operativo, ciò che accade nel calcolatore è che il valore binario corrispondente al numero 12 in base 10 viene scritto nell'area di memoria associata all'identificatore `a`. Nell'istruzione

```
c = a + b
```

invece, viene calcolato il valore corrispondente all'espressione `a + b` e tale valore viene assegnato alla variabile `c`. Ciò che accade è che i valori memorizzati nelle aree di memoria associate agli identificatori `a` e `b` vengono sommati all'interno del processore, dopo aver eventualmente (i valori potrebbero già trovarsi nei registri) trasferito tali valori dalla memoria centrale ai registri del processore. Il valore risultante è poi scritto nell'area di memoria associata all'identificatore `c`.

Tenendo presente il significato dell'operatore =, è facile notare come le istruzioni

```
10 = a
```

oppure

```
a + b = c
```

non siano valide nel linguaggio C, anche se in matematica hanno perfettamente senso. La prima assegna il valore di una variabile ad una costante, mentre la seconda assegnerebbe il valore di una variabile ad una espressione, ma quest'ultima non è associabile a nessuna area di memoria.

Evidentemente il programma presentato non è molto utile, se non per illustrare l'uso delle variabili. Infatti, se si usasse il programma d'esempio per calcolare delle somme, esso andrebbe ricompilato ogni volta che si desidera cambiare i valori da sommare. Una versione più generale del programma potrebbe prevedere di leggere i dati da tastiera, come si vedrà nella Sezione 4.9, oppure in modo ancora più efficiente, leggendo i dati da un file. La lettura da file è illustrata nel Capitolo 12.

4.8 VISUALIZZAZIONE A VIDEO

Nel semplice esempio precedente è da notare come venga utilizzata la funzione `printf` per visualizzare a video il valore della variabile `c`. L'istruzione

```
printf("La loro somma e' %d\n", c);
```

visualizza il valore numerico della *variabile* `c` *di tipo* `int` sostituendo, all'interno della stringa `La loro somma è %d`, la sequenza di caratteri che corrisponde al valore numerico della variabile. Il valore numerico memorizzato nella variabile `c` viene sostituito al posto dei caratteri `%d`. La funzione `printf` utilizza infatti la sequenza di caratteri `%d` per individuare la posizione all'interno della stringa di partenza alla quale inserire il valore numerico. Nel caso specifico, la lettera `d` nella sequenza `%d` informa la funzione `printf` che il tipo della variabile è intero. Il carattere `d` sta infatti per (d)ecimale.

4.9 LETTURA DI DATI DA TASTIERA

Talvolta è necessario richiedere dei dati all'utente per poter effettuare un'elaborazione, che tipicamente li inserisce da tastiera.

Per l'input da tastiera si utilizza la famiglia di funzioni `scanf`, che in generale si occupano di fare il parsing di una stringa di caratteri e di estrarne dei valori, i quali vengono poi assegnati ad opportune variabili[9].

[9]Fare il parsing significa appunto interpretare il contenuto di un insieme di caratteri.

Il semplice programma seguente è la versione un po' più raffinata del programma illustrato in Sezione 4.7, il quale legge due numeri da tastiera, li somma, e visualizza il risultato a video.

Il programma è contenuto nel file `somma_scanf.c`.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int a, b, c;
6
7    printf("Scrivi due numeri interi\n");
8    scanf("%d %d", &a, &b);
9    c = a + b;
10   printf("La loro somma e' %d\n", c);
11
12   return 0;
13 }
```

La chiave della lettura da tastiera risiede nella funzione `scanf`. Nell'esempio, la funzione `scanf` alla linea 8 accetta due numeri interi immessi da tastiera e separati da uno spazio[10]. Una volta immessi i dati e battuto il tasto INVIO, il valore numerico dei dati viene assegnato alle variabili `a` e `b` rispettivamente. Tali variabili possono essere successivamente utilizzate per l'elaborazione, che in questo semplice esempio consiste in una somma.

Per il momento non ci soffermiamo sul modo in cui la `scanf` assegna il valore corretto alle variabili. L'unico aspetto importante è il significato della sequenza di caratteri `%d`, che indica alla `scanf` che i valori introdotti da tastiera sono numeri interi in notazione decimale, quindi non sono nè stringhe di caratteri, nè numeri a virgola mobile, nè interi in notazioni diverse da quella decimale[11].

La e-commerciale è un operatore che ritorna l'indirizzo in memoria del suo operando (le variabili `a` e `b` nell'esempio). La funzione `scanf` richiede che le vengano passati gli indirizzi delle variabili a cui assegnare i valori letti da tastiera. Il passaggio di parametri nel quale è richiesto l'indirizzo dei parametri stessi verrà descritto in dettaglio nella Sezione 8.5.

NOTA

Nella Sezione 15 si raccomanda di evitare la funzione `scanf` per l'input di dati, e viene proposto per questo scopo il metodo basato sulla funzione `fgets` (Sezione 12.6). Negli esempi presentati l'utilizzo della `scanf` viene fatto per motivi didattici, in quanto si ritiene che l'uso di `fgets` richieda dei concetti troppo avanzati per essere introdotti un questi primi esempi.

4.10 SPECIFICA DI FORMATO PER `PRINTF` E `SCANF`

Le funzioni `scanf` e `printf`, ma anche tutte le altre funzioni della stessa famiglia come `fscanf`, `fprintf`, `sscanf` e `sprintf`, accettano una serie di specificatori per il formato delle variabili che devono leggere/scrivere.

La specifica di formato è sempre introdotta dal carattere %, seguito da una lettera che indica il tipo della variabile. Alcuni degli specificatori di formato più utilizzati sono riportati in Tabella 4.1, mentre la trattazione dettagliata dei vari tipi di dato è riportata in Sezione 6.

[10] L'input viene acquisito correttamente anche se viene premuto il tasto INVIO anche dopo aver inserito il primo numero.
[11] Puo' essere interessante verificare cosa succede se, invece che due numeri interi, viene inserita per esempio una stringa di caratteri non numerici

Tabella 4.1 Tabella degli specificatori di formato.

codice	tipo	nota
d	`int`	la `scanf` intero in base 10; carattere di segno opzionale
i	`int`	la `scanf` legge un intero in base 16 se inizia con 0x o 0X, base 8 se inizia con 0, altrimenti base 10; carattere di segno opzionale
o	`unsigned int`	intero senza segno in base 8
u	`unsigned int`	intero senza segno in base 10
x X	`unsigned int`	intero senza segno in base 16
f e g E	`float`	numero in virgola mobile; segno opzionale
s	`*char`	stringa di caratteri diversi dallo spazio
c	`*char`	stringa di caratteri di lunghezza specificata
n	`int`	inserisce nella variabile corrispondente il numero di caratteri letti finora

Nel caso la funzione `scanf` legga una stringa, il vettore di caratteri che deve contenere il risultato deve essere di dimensione adeguata. Inoltre viene aggiunto il carattere '\0', che corrisponde al carattere ASCII di valore numero 0 (zero) alla fine della stringa. Questo carattere ha un significato molto particolare quando viene inserito in una stringa di testo, come verrà chiarito nel capitolo relativo alle stringhe.

Gli specificatori di formato ammettono di essere preceduti dal carattere `l` (elle), che indica che la relativa variabile è di tipo `long` se si tratta di interi o di `double` se si tratta di numeri in virgola mobile.

Inoltre è permesso specificare quante cifre devono essere stampate, per esempio decidendo di stampare un numero in virgola mobile con 2 sole cifre dopo la virgola. Per esempio, le seguenti istruzioni:

```
printf("012345678901234567890123456789\n");
printf("|%4d| |%4d| |%4d|\n", 1, 10, 10000);
printf("|%5.2f| |%7.2f| |%2.1f|\n", 1.0, 10.0, 10000.0);
```

producono il seguente output:

```
012345678901234567890123456789
|   1| |  10| |10000|
| 1.00| |  10.00| |10000.0|
```

La prima istruzione stampa 3 numeri interi, specificando che *il numero minimo* di caratteri da stampare deve essere 4 per ogni numero. Se il numero da stampare è composto da meno di 4 caratteri, vengono aggiunti degli spazi prima del numero stesso.

La seconda istruzione stampa 3 numeri in virgola mobile usando lo specificatore di formato `%f`. Lo specificatore di formato nella forma `%X.Yf` richiede all'istruzione `printf` di stampare almeno X caratteri in tutto, dei quali Y caratteri devono comparire dopo il punto decimale.

Capitolo 5

ISTRUZIONI E STRUTTURE DI CONTROLLO

I costrutti di controllo sono quegli elementi sintattici del linguaggio che permettono di determinare il flusso di esecuzione di un programma. In tal senso, è possibile effettuare la ripetizione di istruzioni sulla base del verificarsi di opportune condizioni (costrutti iterativi), oppure eseguire delle istruzioni invece che altre, sempre sulla base di condizioni logiche che si verificano o meno (costrutti condizionali). Infine, è semplicemente possibile saltare ad una specifica istruzione o parte del programma. Combinando questi costrutti, è possibile mettere in pratica il paradigma della programmazione strutturata. I costrutti che verranno analizzati meglio nelle prossime sezioni sono i seguenti:

```
if ( espr ) istr [ else istr ]
while ( espr ) istr
for ( espr ; espr ; espr ) istr
do istr while ( espr ) ;
switch ( espr-intera ) { case: .... }
break ;
continue ;
return [ espr ] ;
```

La parentesi quadra indica elementi opzionali che possono essere o meno presenti e non fa parte della sintassi del costrutto.

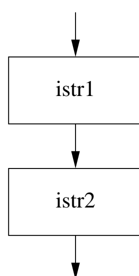


Figura 5.1 Istruzioni composte.

NOTA

Sono corrette le istruzioni

- `return;`
- `return 0;`

non sono corrette le istruzioni

- `return [];`
- `return [0];`

Il termine `istr` può indicare

- un'espressione terminata da punto-e-virgola
- un costrutto di controllo
- un blocco delimitato da parentesi graffe.

Il concetto di *espressione*, indicato con `espr`, include tutto, compresi gli assegnamenti a una variabile, tranne i costrutti di controllo.

Le espressioni sono utilizzate tipicamente per valutare delle condizioni logiche che determinano se continuare ad eseguire un ciclo oppure quale blocco di codice eseguire in un costrutto condizionale. È fondamentale tenere presente che in linguaggio C il valore logico "falso" è rappresentato dal valore numerico 0 (zero), mentre tutti i valori numerici diversi da 0 rappresentano al valore logico "vero".

Per la separazione di parole chiave, espressioni e ogni altro elemento atomico del linguaggio, si possono usare in modo equivalente le andate a capo, gli spazi e i tab. Questa possibilità permette di impaginare il codice sorgente del programma secondo regole stilistiche opportune. Lo stile di impaginazione è quindi libero e programmatori diversi usano stili diversi. È comunque importante non abusare di questa libertà e scrivere codice ordinato e leggibile, facendo rientrare opportunamente i blocchi logici.

5.1 ISTRUZIONI COMPOSTE

Le istruzioni composte sono schematizzate dal diagramma di flusso di Figura 5.1 e sono della forma

```
{ istr1 istr2 }
```

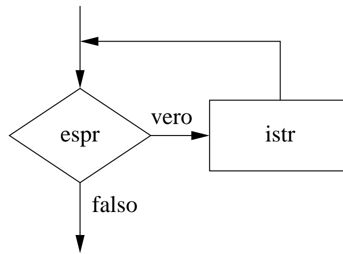


Figura 5.2 Il costrutto `while`.

Sono raggruppate all'interno di parentesi graffe e ciascuna istruzione è terminata dal punto-e-virgola. Queste istruzioni costituiscono un raggruppamento logico di istruzioni diverse. Nella sintassi precedente non è indicato il punto e virgola poiché si assume che una istruzione indicata con `istr` lo includa. Per esempio:

```
{
  s = s + 1.0 / (i * i * i);
  i = i + 1;
}
```

5.1.1 L'operatore virgola

L'operatore "," (virgola) viene anche utilizzato per concatenare la valutazione di espressioni. La sintassi è la seguente:

```
espr1 , espr2
```

ha come effetto la valutazione di entrambe le espressioni. Il risultato dell'espressione complessiva è però dato dal risultato della valutazione della sola `espr2`.

Un esempio generico è il seguente:

```
a = ( espr1 , espr2 ) ;
```

che ha il significato di assegnare ad `a` il valore risultante dal calcolo di `espr2` dopo aver comunque calcolato `espr1`. Per esempio, l'istruzione

```
a = ( b = 0 , 10 * 40 ) ;
```

valuta la prima espressione assegnando 0 alla variabile `b`, e poi calcola il risultato della seconda espressione e lo assegna alla variabile `a`.

5.2 IL COSTRUTTO `WHILE`

Il costrutto `while` serve per realizzare un ciclo (o loop), definito dal diagramma di flusso di Figura 5.2. Assume la forma

```
while ( espr ) istr
```

Il programma è contenuto nel file `somma_while.c`.

Un esempio di utilizzo del costrutto `while` è il seguente:

```
1  #include <stdio.h>
2
3  int main()
4  {
5    long i = 1, max;
6    double somma = 0.0;
7
8    printf("numero di iterazioni: ");
9    scanf("%ld", &max);
10   while (i <= max) {
11     somma = somma + 1.0 / (i * i * i);
12     i = i + 1;
13   }
14   printf("la somma e' %f\n", somma);
15   return 0;
16 }
```

Si noti che le istruzioni che devono essere eseguite ad ogni ciclo sono 2: il calcolo dell'espressione che incrementa il valore della variabile `somma` e l'incrementdo della variabile `i`, rispettivamente alle linee 11 e 12. Per questo motivo le due istruzioni sono racchiuse da parentesi graffe. L'utilizzo delle parentesi graffe specifica al compilatore che intendiamo eseguire *entrambe* le istruzioni ad ogni iterazione.

La caratteristica peculiare di un ciclo realizzato con il costrutto `while` è che il blocco di istruzioni nel ciclo potrebbe non essere *mai* eseguito. Se la condizione è falsa quando viene valutata per la prima volta, allora il ciclo non esegue nemmeno una iterazione. Se per esempio si inserisce un valore di `max` strettamente minore di 1, la condizione è falsa, e quindi non viene effettuata alcuna iterazione.

L'impostazione errata della condizione di terminazione del ciclo può causare vari tipi di errori: la mancata esecuzione del ciclo, l'esecuzione infinita del ciclo, l'esecuzione di un numero errato di cicli. Se si usano gli operatori di disuguaglianza per controllare il ciclo, fare attenzione all'uso dell'operatore ($>$ oppure $>=$ piuttosto che $<$ invece di $<=$), e del valore limite del ciclo. Spesso capita di eseguire un ciclo in più o in meno del necessario a seconda delle scelte fatte.

5.3 IL COSTRUTTO DO-WHILE

Il costrutto `do-while` serve per realizzare un ciclo definito dal diagramma di flusso di Figura 5.3. Assume la forma

```
do istr while ( espr ) ;
```

Il programma è contenuto nel file `somma_do_while.c`.

Per esempio:

```
1  #include <stdio.h>
2
3  int main()
```

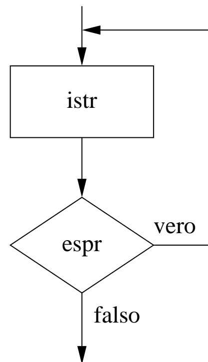


Figura 5.3 Il costrutto `do-while`.

```
4  {
5    long i = 1, max;
6    double somma = 0.0;
7
8    printf("numero di iterazioni: ");
9    scanf("%ld", &max);
10
11   do {
12     somma = somma + 1.0 / (i * i * i);
13     i = i + 1;
14   } while (i < max);
15   printf("La somma e' %f\n", somma);
16   return 0;
17 }
```

A differenza del costrutto `while` il blocco di istruzioni nel ciclo *viene sempre eseguito almeno una volta*. Infatti la condizione che controlla l'esecuzione del ciclo viene controllata *alla fine* del ciclo.

5.4 IL COSTRUTTO FOR

Il costrutto `for` serve per realizzare un ciclo definito dal diagramma di flusso di Figura 5.4. Assume la forma

```
for ( espr1 ; espr2 ; espr3 ) istr
```

Il programma è contenuto nel file `somma_for_inc.c`.

Un esempio è il seguente

```
1  #include <stdio.h>
2
3  int main()
4  {
5    long i, max;
```

Figura 5.4 Il costrutto `for`.

```
6     double somma;
7
8     printf("valore massimo di i: ");
9     scanf("%ld", &max);
10
11    for(i = 1, somma = 0.0; i <= max; i = i + 1)
12      somma = somma + 1.0 / (i * i * i);
13
14    printf("la somma e' %f\n", somma);
15    return 0;
16  }
```

La variabile `i`, che regola il numero di iterazioni, è solitamente chiamata *contatore*. Si noti l'utilizzo dell'operatore virgola per permettere l'inizializzazione di 2 variabili prima dell'inizio del ciclo.

Il funzionamento del programma è il seguente: viene inizializzata la variabile `i` pari al valore 1 e la variabile `somma` pari al valore 0.0; ad ogni ciclo viene calcolata l'espressione che incrementa il valore di `somma`, e successivamente viene incrementato di 1 il valore del contatore `i`. Il ciclo continua fintanto che la condizione `i <= max` risulta vera, quindi termina quando `i` diviene maggiore di `max`.

Si noti che, al contrario degli esempi precedenti, il *corpo* del ciclo `for` è costituito da una sola istruzione, alla linea 12. Per questo motivo non è necessario includere tale istruzione tra parentesi graffe. Il comportamento del programma sarebbe comunque lo stesso nel caso in cui l'istruzione di somma fosse inclusa tra parentesi graffe come segue:

```
for(i = 1, somma = 0.0; i <= max; i = i + 1) {
  somma = somma + 1.0 / (i * i * i);
}
```

Questo fatto viene sottolienato in quanto spesso capita che si sbagli nel comprendere quali sono le istruzioni eseguite da un ciclo di questo tipo per il fatto che le istruzioni da eseguire non sono delimitare da parentesi graffe.

NOTA Generalmente, l'espressione `espr2` usata nel test di uscita del ciclo `for`, quando ci sia da effettuare un numero fisso di iterazioni (es., iterando `max` volte partendo da 0), è bene che sia del tipo "`i < max`", con il minore stretto. Questo perché, in questo caso, `max` indica direttamente il numero di iterazioni.

È sintatticamente corretto scrivere un'istruzione come segue

```
for(i = 1, somma = 0.0; i <= max; i = i + 1) ;
```

nella quale subito dopo il ciclo `for` compare il punto-e-virgola. In questo caso il corpo del ciclo è costituito da 0 istruzioni, quindi viene effettuato il ciclo nel quale si eseguono soltanto le istruzioni necessarie a gestire il ciclo stesso (incremento di contatori e valutazione delle condizioni).
L'errore che spesso si commette è per esempio il seguente:

```
for(i = 1, somma = 0.0; i <= max; i = i + 1) ;
printf("%d\n", i);
```

che nelle intenzioni del programmatore dovrebbe far stampare il valore di `i` *ad ogni iterazione*. A causa dell'erronea presenza del punto-e-virgola dopo il ciclo `for`, il ciclo viene eseguito senza effettuare alcuna stampa, e soltanto al termine del ciclo ha luogo l'operazione di stampa, che quindi viene effettuata *una volta sola*.

Il programma è contenuto nel file `somma_for_dec.c`.

Un altro esempio, nel quale il contatore viene fatto decrementare, è il seguente:

```
1  #include <stdio.h>
2
3  int main()
4  {
5    long i, max;
6    double somma;
7
8    printf("numero di iterazioni: ");
9    scanf("%ld", &max);
10
11   for(i = max, somma = 0; i > 0; i = i - 1)
12     somma = somma + 1.0 / (i * i * i);
13
14   printf("%f\n", somma);
15   return 0;
16 }
```

In questo secondo esempio, il quale permette di calcolare il valore di `somma` correttamente quanto l'esempio precedente, il contatore viene inizializzato al valore di `max`, e poi fatto decrementare di una unità ad ogni ciclo, fintanto che il contatore stesso è strettamente maggiore di 0.

NOTA — Gli identificatori `i` e `j` sono spesso usati per le variabili che fungono da contatori. A rischio di essere ovvio, si noti che può essere usata un identificatore qualsiasi.

Utilizzare la virgola invece del punto-e-virgola per separare le espressioni in un costrutto `for` costituisce un errore di sintassi.

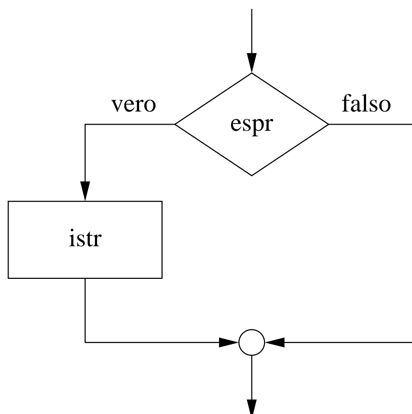


Figura 5.5 Il costrutto `if`.

5.5 IL COSTRUTTO IF

Il costrutto `if` serve per realizzare l'istruzione di salto condizionale specificata dal diagramma di flusso di Figura 5.5. Assume la forma

```
if ( espr ) istr
```

Il programma è contenuto nel file `max_if.c`.

Un semplice esempio che restituisce il valore massimo determinato sui due valori che vengono inseriti da tastiera, è il seguente:

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int a, b, max;
6
7    scanf("%d %d", &a, &b);
8    max = a;
9    if (b > a)
10     max = b;
11   printf("il massimo e': %d", max);
12
13   return 0;
14 }
```

Il costrutto `if` ammette l'enunciato opzionale `else`, per cui il costrutto `if-else` serve per realizzare l'istruzione condizionale definita dal diagramma di flusso di Figura 5.6. Assume la forma

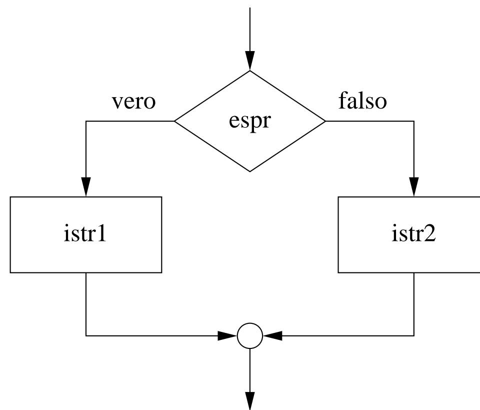


Figura 5.6 Il costrutto `if-else`.

```
if ( espr ) istr1 else istr2
```

Il programma è contenuto nel file `max_if_else.c`.

Nell'esempio seguente il costrutto `if-else` è utilizzato per determinare il massimo tra due numeri interi.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int a, b, max;
6
7    scanf("%d %d", &a, &b);
8    if (b > a) max = b;
9    else max = a;
10   printf("il massimo e': %d", max);
11
12   return 0;
13 }
```

5.6 IL COSTRUTTO SWITCH

Il costrutto di controllo `switch` serve a scegliere tra diversi blocchi di istruzioni in base al valore di una espressione intera. La sintassi è diversa da quella degli altri costrutti di controllo, perché le parentesi graffe sono obbligatorie .

La sintassi completa è la seguente:

```
switch ( espressione-intera ) {
  case espressione-costante :
    [ istr ]
```

```
    [ ... ]
    [ break ; ]
  case espressione-costante :
    [ istr ]
    [ ... ]
    [ break ; ]
  [ default: ]
    [ istr ]
    [ ... ]
    [ break ; ]
}
```

Anche in questo caso, le parentesi quadre indicano parti del costrutto che sono opzionali.

Le espressioni di ogni `case` devono essere espressioni intere e costanti, cioè valutabili all'atto della compilazione. La presenza di istruzioni dopo ogni `case` è facoltativa, per permettere di raggruppare lo stesso codice in relazione a diversi casi.

> *NOTA* Un carattere tra apici, cioè una costante di tipo `char`, è un numero intero, quindi può essere utilizzata all'interno di una clausola `case`.

La presenza di `break` alla fine di un caso è facoltativa. La mancanza di un `break` determina l'esecuzione del codice associato al caso successivo.

> *NOTA* Dal momento che tipicamente il `break` è presente in ogni clausola `case`, quando lo si omette è buona norma commentarne la mancanza, perché non sembri una dimenticanza a chi legge il codice.

La clausola `default` è facoltativa; se presente viene selezionata quando l'espressione del costrutto `switch` non trova corrispondenza tra i casi elencati. Non è obbligatorio che `default` sia l'ultimo caso del costrutto.

Nell'esempio seguente il costrutto `switch` viene usato per selezionare un comando specifico sulla base dell'input dell'utente. Le operazioni eseguite si limitano a stampare dei messaggi che servono a verificare quale parte del codice viene eseguita. In un programma completo, le istruzioni di stampa verrebbero sostituite, o completate, con delle chiamate a funzioni o con istruzioni addizionali che effettivamente eseguano le operazioni desiderate.

Il programma è contenuto nel file `switch.c`.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    char opt;
6
7    printf("inserire il codice del comando (r, w, x): ");
8    scanf("%c", &opt);
9
10   switch(opt) {
11     case 'r' :
12     case 'R' :
13       printf("lettura\n");
```

```
14          break;
15        case 'x' :
16        case 'X' :
17          printf("esecuzione\n");
18          /* poi la scrittura */
19        case 'w' :
20        case 'W' :
21          printf("scrittura\n");
22          break;
23        default:
24          printf("operazione non valida\n");
25          return -1;
26      }
27      return 0;
28    }
```

L'utente può inserire i caratteri `rwx` o `RWX`; i dati vengono letti con la `scanf` (linea 8) che corrispondono alle seguenti operazioni:

- r – lettura
- w – scrittura
- x – esecuzione

Nel caso si richieda una esecuzione, le operazioni che effettivamente devono essere eseguite sono l'esecuzione e la scrittura. Nell'esempio si possono notare i seguenti accorgimenti:

- per gestire caratteri maiuscoli e minuscoli, sono presenti due clausole `case` per ciascun possibile comando (per esempio, alle linee 11 e 12), che portano all'esecuzione delle stesse istruzioni poiché non c'è l'istruzione `break` che esce dal costrutto;
- l'uso dell'istruzione `break` per uscire dal costrutto `switch` nel caso di lettura (linea 14);
- l'uso dell'istruzione `break` per uscire dal costrutto `switch` nel caso di sola scrittura (linea 22);
- nel caso dell'opzione `x`, non si usa la `break`, in modo che, una volta eseguite le istruzioni relative a tale opzione, vengano eseguite anche le istruzioni successive relative alla scrittura (poi si incontra `break`);
- la clausola `default` (linea 23) viene usata per gestire le condizioni di errore, ovvero quando viene introdotto un carattere non previsto.

Normalmente `switch` viene usato per selezionare tra diversi comandi, oppure nella valutazione dei parametri sulla riga di comando. L'uso di due o più `case` per lo stesso blocco di codice è raro .

5.7 LE ISTRUZIONI BREAK E CONTINUE

Le istruzioni `break` e `continue` vengono utilizzate per controllare il flusso di esecuzione all'interno di cicli `while`, `do-while` e `for`.

Il comportamento delle due istruzioni è il seguente:

- `break` termina immediatamente il ciclo più interno nel quale è contenuta;

- `continue` passa immediatamente al punto in cui viene effettuato il test del loop

Il programma è contenuto nel file `break_continue.c`.

L'esempio seguente

```
1  do {
2    printf("valore di n: ");
3    scanf("%d", &n);
4
5    if (n < 0) {
6      printf("  %d < 0: uscita dal loop...\n", n);
7      break;
8    }
9
10   if (n > 10) {
11     printf("  %d > 10: non ammesso\n", n);
12     continue;
13   }
14
15   /* utilizza il valore di n */
16   printf("  valore immesso: n = %d\n", n);
17
18 } while (1);
```

è costituito da un cosiddetto *ciclo infinito*: la condizione valutata nella `while` alla linea 18 è sempre vera. Si ricordi infatti che la condizione falsa si ha solo quando il valore dell'espressione è pari a 0, mentre altrimenti è considerata vera. Da notare che, in questo caso, il ciclo infinito non costituisce un errore di programmazione, come può accadere in altri casi, ma è stato realizzato appositamente tenendo conto della possibilità di uscire dal ciclo per mezzo dell'istruzione `break` alla linea 7. Se viene immesso un valore negativo per la variabile `n`, allora il ciclo `do-while` viene subito terminato, mentre se viene inserito un numero maggiore di 10, si salta all'iterazione successiva per effetto dell'istruzione `continue` presente in 12. Il risultato è che verranno stampati a video i soli valori inseriti che siano compresi nell'intervallo $[0, 9]$.

È da notare come, nel semplice esempio proposto, sarebbe molto facile gestire il flusso del programma all'interno del ciclo anche facendo a meno delle istruzioni `break` e `continue`. Quando ciò è possibile, è bene farlo, in quanto le istruzioni `break` e `continue` possono rendere più complicata la lettura di un programma complesso.

5.8 IL COSTRUTTO GOTO

L'istruzione `goto` esiste anche nel linguaggio C[12]. Il suo utilizzo è sconsigliato in quanto, in genere, complica la comprensione del programma, perché rende difficile seguirne il flusso logico. Inoltre qualsiasi algoritmo può essere implementato usando i costrutti illustrati precedentemente senza l'utilizzo della `goto`, e quindi la disponibilità della `goto` è generalmente superflua.

Vi sono però delle eccezioni nelle quali l'utilizzo della `goto` può rivelarsi utile, per esempio quando si deve uscire da cicli più volte annidati a causa di situazioni anomale (errori). In pratica, la `goto`

[12] È un costrutto utilizzato moltissimo in altri linguaggi, come il BASIC.

viene utilizzata in casistiche standard (gestione errori) quando il suo utilizzo rende più semplice il programma di quanto non lo sarebbe usando altri costrutti.

La sintassi del comando è:

```
goto <etichetta> ;
```

Nel programma vi deve essere una istruzione etichettata con lo stesso nome secondo la sintassi:

```
<etichetta>: <istruzione>
```

Il funzionamento è molto semplice: quando viene incontrata l'istruzione `goto`, viene effettuato un salto incondizionato al punto del programma al quale è posta l'etichetta.

Un esempio è il seguente

Il programma completo è contenuto nel file `goto.c`.

```
1  while (1) {
2    /* .... */
3    if (q) break;
4    while (1) {
5      /* .... */
6      /* esce solo dal ciclo piu' interno */
7      if (p) break;
8      /* .... */
9      /* esce dal ciclo piu' esterno */
10     if (errore) goto uscita;
11     /* .... */
12   }
13 }
14
15 uscita:
16 if (errore) { /* gestione del problema */  }
```

L'esempio sopra riportato consiste di due cicli annidati, alle linee 1 e 4. Entrambi sono cicli infiniti, in quanto la condizione che comporta la continuazione del ciclo è sempre vera. Sono però state predisposte tre condizioni di uscita, nel caso si verifichino le condizioni associate alle variabili `q`, `p` ed `errore`, rispettivamente alle linee 3, 7 e 10.

In caso si verifichi la condizione `q`, viene utilizzata l'istruzione `break` per uscire dal ciclo corrente (il più esterno), e quindi terminare le iterazioni. Se si verifica la condizione `p` viene usata sempre l'istruzione `break` per uscire dal ciclo corrente, ma in questo caso viene terminato solo il ciclo più interno. In caso di errore, invece, è necessario uscire direttamente dal ciclo interno, ma ciò è possibile soltanto utilizzando l'istruzione `goto`, che in questo caso salta alla linea 15, indicata con l'etichetta `uscita`, e da questo punto riprende la normale esecuzione del programma.

Capitolo 6

TIPI DI DATI

In un linguaggio di programmazione, un tipo di dato identifica l'insieme di valori che possono essere assunti da una variabile o da una espressione, e le operazioni che su di essi si possono svolgere.

Il C è un linguaggio cosiddetto *tipizzato*, ovvero richiede che sia esplicitamente associato un tipo a ciascun dato utilizzato nel programma. La tipizzazione permette di effettuare delle verifiche sull'uso dei dati, in modo da controllare la coerenza dei tipi coinvolti e di segnalare in modo automatico al programmatore eventuali anomalie. Per esempio, non sarà possibile assegnare una stringa di testo ad una variabile in virgola mobile.

Per questo motivo è necessario *dichiarare* il tipo delle variabili e altri dati (costanti, valori di ritorno delle funzioni, ecc.), in modo da informare esplicitamente il compilatore su quale tipo di dato sia associato alla variabile che si intende utilizzare.

La dichiarazione del tipo di dato permette anche al compilatore di gestire l'allocazione in memoria dello spazio necessario a memorizzare la variabile. Infatti a ciascun tipo di dato è associata una certa dimensione in byte. Quando una variabile viene dichiarata di un determinato tipo, il compilatore si preoccuperà di riservare la corrispondente quantità di memoria per la sua memorizzazione.

I *dati semplici* sono i tipi di dato primitivi definiti dal linguaggio, come numeri interi, in virgola mobile, o puntatori. Consideriamo invece *dati composti* i tipi di dato che aggregano tipi di dati semplici, come gli array, le strutture dati, le unioni e i tipi definiti dall'utente. Nel linguaggio originale non esiste il tipo *boolean*, che rappresenta un valore vero o falso, che invece è presente in modo "nativo" in vari altri linguaggi. Se un valore è zero viene considerato falso, se è diverso da zero viene considerato vero. D'altra parte, nell'ultima versione standardizzata del linguaggio C, il C99, è stato introdotto il tipo `bool` proprio per rappresentare i valori logici vero/falso.

In questa sezione verranno descritti in dettagli i tipi di dati nativi disponibili nel linguaggio C.

Tabella 6.1 I tipi interi previsti nel linguaggio C.

	con segno	senza segno	dim. minima (bit)	dim. tipica (bit)
char	signed char	unsigned char	8	8
short	signed short	unsigned short	16	16
int	signed int	unsigned int	16	32
long	signed long	unsigned long	32	64

6.1 TIPI DI DATO E MEMORIA

Per comprendere pienamente e saper utilizzare correttamente costanti, variabili, e qualsiasi altro aspetto del linguaggio che ha attinenza con i tipi di dati è fondamentale avere sempre presente *come viene gestita la memoria* quando si opera su un determinato oggetto del linguaggio.

Quando viene memorizzato un valore in memoria, di qualsiasi tipo esso sia, le informazioni fondamentali per permettere all'elaboratore di gestire correttamente l'informazione memorizzata sono tre:

1. il punto della memoria (indirizzo) al quale l'informazione viene memorizzata
2. da quanti byte è costituita tale informazione
3. qual è il tipo di dato che essa rappresenta

In realtà, dal momento che a ciascun tipo di dato è associata una dimensione ben definita, il numero di byte che costituiscono l'informazione memorizzata è ricavabile dal tipo.

Per quanto riguarda il tipo di dato, questo *permette di interpretare in modo consistente i bit* immagazzinati all'indirizzo al quale l'informazione è memorizzata. Infatti, a parità di numero e valore dei bit memorizzati ad un certo indirizzo, tali bit possono essere interpretati in modo diverso a seconda che rappresentino un numero intero, un numero in virgola mobile, una stringa o qualsiasi altro tipo di dato. È quindi l'associazione tra tipo di dato e valori memorizzati ad un certo indirizzo in memoria che ci permettono di risalire al valore corretto rappresentato dai bit memorizzati.

6.2 TIPI INTERI

I tipi interi predefiniti del linguaggio sono riportati in Tabella 6.1, insieme alle dimensioni predefinite di ciascun tipo.

Gli interi `signed` hanno la possibilità di memorizzare dei numeri con segno, mentre gli interi `unsigned` indicano interi senza segno. Normalmente `signed` non si usa perché è il comportamento predefinito.

Dal momento che è necessario utilizzare un bit in più per rappresentare il segno nel caso di interi `signed`, questo cambia l'intervallo nel quale possono variare i valori memorizzati. Per esempio, una variabile di tipo `char` può assumere valori nell'intervallo $-128 \ldots 127$, cioè da -2^7 a $2^7 - 1$ (si noti l'esponente 7: il tipo `char` è a 8 bit, ma 1 viene usato per il segno) se è di tipo `signed`, ma se di tipo `unsigned char` l'intervallo diventa $0 \ldots 255$, cioè fino a $2^8 - 1$. Gli intervalli dei possibili valori per i tipi interi del C, sulla base della dimensione del dato, sono riportati in Tabella 6.2.

Sulla lunghezza (dimensione in bit) di tali tipi non si possono fare assunzioni, poiché essa può dipendere dall'architettura per la quale il programma viene compilato e dal compilatore stesso. In pratica, però, è garantito che il tipo `char` sia sempre di 8 bit.

La dimensione esatta del tipo `int` dipende quindi dal processore ospite. Generalmente è la dimensione della parola sull'hardware utilizzato. Questo significa che l'intero può essere a 16, 32,

Tabella 6.2 Intervalli dei valori possibili per i diversi tipi interi del C.

n. bit	segno	min	max
8	signed	-128	+127
16	signed	-32.768	+32.767
32	signed	-2.147.483.648	+2.147.483.647
8	unsigned	0	+255
16	unsigned	0	+65.535
32	unsigned	0	+4.294.967.295

64 bit in base alla macchina in uso[13]. Viene comunque garantito che una variabile di tipo `int` possa utilizzare almeno 16 bit.

6.2.1 Costanti di tipo intero

Per quanto riguarda le costanti di tipo intero, si ha che

- sono sequenze di caratteri numerici senza il punto decimale;
- se la prima cifra è 0 il numero è interpretato come scritto secondo la notazione ottale
- se la prima cifra è 0 seguito da x (X) il numero è considerato esadecimale

Il numero intero decimale 127 può perciò essere scritto sia come 0177 che 0x7f.

- le costanti di tipo `long` hanno come ultimo carattere L (l)
- costanti di tipo `unsigned` U (u)
- costanti di tipo `char` possono essere anche scritte come un carattere tra apici

Se il calcolatore utilizza la codifica ASCII, le costanti di tipo `char` che valgono '1', 49, 0x31, 061 sono rappresentazioni diverse dello stesso valore.

NOTA — Il tipo `char` rappresenta un intero a 8 bit, e nella codifica ASCII i caratteri si rappresentano con codici a 8 bit. Ecco perché il carattere '1' equivale al valore numerico 49 decimale.

La Figura 6.1 riporta l'elenco completo dei caratteri ASCII stampabili, con i relativi codici numerici in notazione decimale, ottale ed esadecimale.

6.3 TIPI A VIRGOLA MOBILE

La specifica delle caratteristiche dei tipi a virgola mobile è soggetta ad uno standard internazionale, l'IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985).

Un numero a virgola mobile è codificato come segue:

- un bit di segno S
- un campo di esponente E

[13] Il numero di bit è generalmente associato alla dimensione del bus dati del processore.

<table>
<tr><td></td><td>32</td><td>040</td><td>0x20</td><td>@</td><td>64</td><td>0100</td><td>0x40</td><td>`</td><td>96</td><td>0140</td><td>0x60</td></tr>
<tr><td>!</td><td>33</td><td>041</td><td>0x21</td><td>A</td><td>65</td><td>0101</td><td>0x41</td><td>a</td><td>97</td><td>0141</td><td>0x61</td></tr>
<tr><td>"</td><td>34</td><td>042</td><td>0x22</td><td>B</td><td>66</td><td>0102</td><td>0x42</td><td>b</td><td>98</td><td>0142</td><td>0x62</td></tr>
<tr><td>#</td><td>35</td><td>043</td><td>0x23</td><td>C</td><td>67</td><td>0103</td><td>0x43</td><td>c</td><td>99</td><td>0143</td><td>0x63</td></tr>
<tr><td>$</td><td>36</td><td>044</td><td>0x24</td><td>D</td><td>68</td><td>0104</td><td>0x44</td><td>d</td><td>100</td><td>0144</td><td>0x64</td></tr>
<tr><td>%</td><td>37</td><td>045</td><td>0x25</td><td>E</td><td>69</td><td>0105</td><td>0x45</td><td>e</td><td>101</td><td>0145</td><td>0x65</td></tr>
<tr><td>&</td><td>38</td><td>046</td><td>0x26</td><td>F</td><td>70</td><td>0106</td><td>0x46</td><td>f</td><td>102</td><td>0146</td><td>0x66</td></tr>
<tr><td>’</td><td>39</td><td>047</td><td>0x27</td><td>G</td><td>71</td><td>0107</td><td>0x47</td><td>g</td><td>103</td><td>0147</td><td>0x67</td></tr>
<tr><td>(</td><td>40</td><td>050</td><td>0x28</td><td>H</td><td>72</td><td>0110</td><td>0x48</td><td>h</td><td>104</td><td>0150</td><td>0x68</td></tr>
<tr><td>)</td><td>41</td><td>051</td><td>0x29</td><td>I</td><td>73</td><td>0111</td><td>0x49</td><td>i</td><td>105</td><td>0151</td><td>0x69</td></tr>
<tr><td>*</td><td>42</td><td>052</td><td>0x2a</td><td>J</td><td>74</td><td>0112</td><td>0x4a</td><td>j</td><td>106</td><td>0152</td><td>0x6a</td></tr>
<tr><td>+</td><td>43</td><td>053</td><td>0x2b</td><td>K</td><td>75</td><td>0113</td><td>0x4b</td><td>k</td><td>107</td><td>0153</td><td>0x6b</td></tr>
<tr><td>,</td><td>44</td><td>054</td><td>0x2c</td><td>L</td><td>76</td><td>0114</td><td>0x4c</td><td>l</td><td>108</td><td>0154</td><td>0x6c</td></tr>
<tr><td>-</td><td>45</td><td>055</td><td>0x2d</td><td>M</td><td>77</td><td>0115</td><td>0x4d</td><td>m</td><td>109</td><td>0155</td><td>0x6d</td></tr>
<tr><td>.</td><td>46</td><td>056</td><td>0x2e</td><td>N</td><td>78</td><td>0116</td><td>0x4e</td><td>n</td><td>110</td><td>0156</td><td>0x6e</td></tr>
<tr><td>/</td><td>47</td><td>057</td><td>0x2f</td><td>O</td><td>79</td><td>0117</td><td>0x4f</td><td>o</td><td>111</td><td>0157</td><td>0x6f</td></tr>
<tr><td>0</td><td>48</td><td>060</td><td>0x30</td><td>P</td><td>80</td><td>0120</td><td>0x50</td><td>p</td><td>112</td><td>0160</td><td>0x70</td></tr>
<tr><td>1</td><td>49</td><td>061</td><td>0x31</td><td>Q</td><td>81</td><td>0121</td><td>0x51</td><td>q</td><td>113</td><td>0161</td><td>0x71</td></tr>
<tr><td>2</td><td>50</td><td>062</td><td>0x32</td><td>R</td><td>82</td><td>0122</td><td>0x52</td><td>r</td><td>114</td><td>0162</td><td>0x72</td></tr>
<tr><td>3</td><td>51</td><td>063</td><td>0x33</td><td>S</td><td>83</td><td>0123</td><td>0x53</td><td>s</td><td>115</td><td>0163</td><td>0x73</td></tr>
<tr><td>4</td><td>52</td><td>064</td><td>0x34</td><td>T</td><td>84</td><td>0124</td><td>0x54</td><td>t</td><td>116</td><td>0164</td><td>0x74</td></tr>
<tr><td>5</td><td>53</td><td>065</td><td>0x35</td><td>U</td><td>85</td><td>0125</td><td>0x55</td><td>u</td><td>117</td><td>0165</td><td>0x75</td></tr>
<tr><td>6</td><td>54</td><td>066</td><td>0x36</td><td>V</td><td>86</td><td>0126</td><td>0x56</td><td>v</td><td>118</td><td>0166</td><td>0x76</td></tr>
<tr><td>7</td><td>55</td><td>067</td><td>0x37</td><td>W</td><td>87</td><td>0127</td><td>0x57</td><td>w</td><td>119</td><td>0167</td><td>0x77</td></tr>
<tr><td>8</td><td>56</td><td>070</td><td>0x38</td><td>X</td><td>88</td><td>0130</td><td>0x58</td><td>x</td><td>120</td><td>0170</td><td>0x78</td></tr>
<tr><td>9</td><td>57</td><td>071</td><td>0x39</td><td>Y</td><td>89</td><td>0131</td><td>0x59</td><td>y</td><td>121</td><td>0171</td><td>0x79</td></tr>
<tr><td>:</td><td>58</td><td>072</td><td>0x3a</td><td>Z</td><td>90</td><td>0132</td><td>0x5a</td><td>z</td><td>122</td><td>0172</td><td>0x7a</td></tr>
<tr><td>;</td><td>59</td><td>073</td><td>0x3b</td><td>[</td><td>91</td><td>0133</td><td>0x5b</td><td>{</td><td>123</td><td>0173</td><td>0x7b</td></tr>
<tr><td><</td><td>60</td><td>074</td><td>0x3c</td><td>\</td><td>92</td><td>0134</td><td>0x5c</td><td>|</td><td>124</td><td>0174</td><td>0x7c</td></tr>
<tr><td>=</td><td>61</td><td>075</td><td>0x3d</td><td>]</td><td>93</td><td>0135</td><td>0x5d</td><td>}</td><td>125</td><td>0175</td><td>0x7d</td></tr>
<tr><td>></td><td>62</td><td>076</td><td>0x3e</td><td>^</td><td>94</td><td>0136</td><td>0x5e</td><td>~</td><td>126</td><td>0176</td><td>0x7e</td></tr>
<tr><td>?</td><td>63</td><td>077</td><td>0x3f</td><td>_</td><td>95</td><td>0137</td><td>0x5f</td><td></td><td>127</td><td>0177</td><td>0x7f</td></tr>
</table>

Figura 6.1 Tabella ASCII.

Tabella 6.3 I tipi a virgola mobile previsti nel linguaggio C.

tipo	significato
float	precisione semplice
double	precisione doppia
long double	precisione estesa

- un campo di mantissa M

A partire da questa codifica, il valore effettivo del numero viene calcolato come segue:

$$(-1)^S \cdot 2^E \cdot M$$

A seconda del numero di bit dedicati a ciascun campo E ed M si possono distinguere tipi floating point caratterizzati da diversa precisione. I tipi a virgola mobile disponibili in C sono elencati in Tabella 6.3

Le costanti a virgola mobile hanno le seguenti caratteristiche:

- i tipi `double` sono caratterizzate o dal punto decimale o dall'esponente (es. 133.3 o 78e-5)
- i tipi `float` sono caratterizzate dal suffisso f(F) (es. 133.3f o 78e-5F)
- i tipi `long double` sono caratterizzate dal suffisso l(L) (es. 133.3l o 78e-5L)

NOTA: Non utilizzare mai l'operatore == per effettuare confronti tra valori in virgola mobile. A causa degli arrotondamenti con cui vengono memorizzati, i valori possono differire dal valore atteso per qualche decimale, e far fallire confronti di uguaglianza.

Ad esempio, se `raggio` è una variabile di tipo `float`, per testare se la variabile vale, diciamo 10.0, la seguente istruzione potrebbe fallire

```
float raggio;

if (raggio == 10.0) {
  /* ... */
}
```

ed è quindi meglio effettuare un test del tipo:

```
float raggio;
float epsilon = 0.00001;

if (fabs(raggio - 10.0) <= epsilon * fabs(raggio)) {
  /* ... */
}
```

che utilizza la funzione di libreria `fabs` la quale calcola il valore assoluto di un numero a virgola mobile.

6.4 I PUNTATORI

Un puntatore memorizza l'*indirizzo in memoria* di una variabile.

Un puntatore si definisce scrivendo il tipo cui si punta, l'asterisco e il nome della variabile. Per esempio

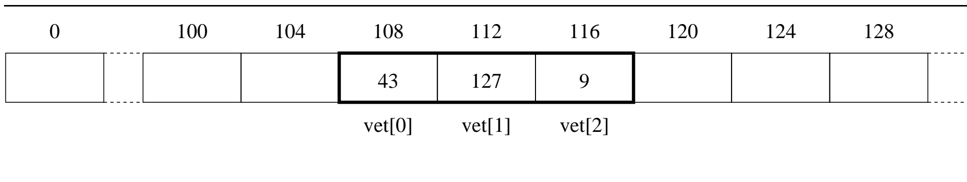


Figura 6.2 Esempio di allocazione in memoria di un vettore: il vettore `vet` è formato da 3 elementi interi.

```
int *p;
```

Conviene leggere il carattere asterisco come *il puntato da*. Nel caso precedente quindi si legge *è intero il [valore] puntato da p*.

Un puntatore, quindi, *non contiene il valore di una variabile*. Nell'esempio precedente, la variabile `p` non contiene un numero intero, ma contiene *l'indirizzo di una locazione in memoria* nella quale saranno immagazzinati un certo numero di bit *da interpretare come un numero intero*.

I puntatori sono tutti della stessa dimensione, e sono a 32 o 64 bit a seconda del processore su cui si lavora. Su tutte le piattaforme un `unsigned long` e un puntatore hanno la stessa dimensione.

Dal momento che i puntatori costituiscono un aspetto chiave e talvolta un po' ostico del linguaggio C, verranno ripresi in modo più dettagliato al Capitolo 7.

6.5 GLI ARRAY

Gli array non sono dei veri e propri tipi nativi del linguaggio C. Piuttosto, essi permettono di memorizzare in *aree contigue di memoria* un certo numero di elementi di *tipo omogeneo*. A seconda della *dimensione* dell'array, essi si distinguono nella pratica in: vettori, matrici, array multi-dimensionali.

È importante sottolineare come per padroneggiare appieno il concetto di array, è indispensabile avere ben presente come viene organizzata la memoria per immagazzinare l'array stesso.

6.5.1 I vettori

I vettori, o array monodimensionali, permettono di allocare un insieme di elementi dello stesso tipo in zone contigue della memoria. La sintassi per la dichiarazione di un vettore è la seguente:

```
nome-tipo identificatore [ cardinalita' ] ;
```

dove

- *nome-tipo* è un qualsiasi tipo di dato, sia semplice che composto
- *identificatore* è il nome che identifica il vettore
- *cardinalità* è un numero intero che indica di quanti elementi è costituito il vettore

Per esempio

```
int vet[10];
```

definisce il vettore *vet* di 10 interi. Il vettore è indicizzato da 0 a 9, ovvero il primo elemento è `vet[0]` mentre l'ultimo è `vet[9]`.

Nell'esempio di Figura 6.2 viene allocato invece un vettore di 3 elementi di tipo intero, ciascuno dei quali occupa 4 byte di memoria. Il vettore è allocato a partire dall'indirizzo 108 e, come si

Utilizzare le parentesi tonde per indicizzare gli elementi di un vettore invece delle parentesi quadre costituisce un errore di sintassi.

può notare, la memoria è allocata in modo contiguo. Il vettore può essere indicizzato nell'intervallo $[0\dots2]$, e nei tre elementi sono memorizzati rispettivamente i valori 43, 127 e 9.

Il programma seguente effettua la somma di numeri interi che, invece di essere memorizzati in variabili singole, vengono memorizzati come elementi di un vettore opportunamente dimensionato. Il programma fornisce un esempio di accesso agli elementi del vettore, sia indicizzando il vettore stesso mendiante una variabile all'interno di un ciclo `for`, che indicizzando uno specifico elemento del vettore con un indice costante.

Il programma è contenuto nel file `vettore.c`.

```
1  #include <stdio.h>
2
3  #define NUM (3)
4
5  int main()
6  {
7    int i, somma;
8    int vet[3];
9
10   printf("inserire i valori da sommare\n");
11   for (i = 0; i < NUM; i++) {
12     printf("vet[%d] = ", i);
13     scanf("%d", &vet[i]);
14   }
15
16   for (i = 0, somma = 0; i < NUM; i++)
17     somma = somma + vet[i];
18
19   printf("somma complessiva: %d\n", somma);
20   printf("somma del primo e ultimo valore: %d\n", vet[0] + vet[NUM - 1]);
21
22   return 0;
23 }
```

Il programma, con il ciclo `for` alla linea 11, legge `NUM` valori da console e li memorizza nel vettore `vet`, dove `NUM` è specificato con la direttiva `#define` alla linea 3. Una volta letti, i valori vengono sommati con il ciclo alla linea 16. Si noti che il valore di `somma` viene inizializzato a 0 contestualmente all'istruzione che specifica il ciclo. Non inizializzare il valore di `somma` avrebbe prodotto una somma inconsistente, dal momento che il valore iniziale della variabile è aleatorio.

Nel caso in cui il vettore venga indicizzato con un indice al di fuori dell'intervallo ammesso, nel migliore dei casi si ha un errore di "segmentation fault", mentre in altri casi si va a scrivere/leggere una variabile che appartiene al programma, allocata subito dopo il vettore. Nel secondo caso, è difficile sapere a priori di che variabile si tratta, quindi questo è da considerarsi un comportamento errato e molto pericoloso del programma, in quanto possono verificarsi comportamenti indesiderati e imprevedibili, e spesso molto difficili da diagnosticare.

Nell'esempio di Figura 6.2, un accesso all'elemento `vet[3]`, che utilizza cioè un indice al di fuori dall'intervallo corretto di indicizzazione del vettore, comporterebbe un accesso ai 4 byte che sono posizionati in memoria all'indirizzo 120, e che non sono allocati per il vettore. Non è detto che a tale indirizzo sia memorizzato un intero, ma dal momento che vi accediamo utilizzando l'indicizzazione del vettore `vet`, che è un vettore di interi, allora i 4 byte presenti a tale indirizzo sono interpretati come un numero intero. Ora, se tale locazione di memoria non è accessibile dal programma, allora sarè generato un errore di "segmentation fault". Se invece la locazione di memoria è accessibile al programma, può darsi che a tale indirizzo sia memorizzata un'altra variabile (a priori non abbiamo la possibilità di sapere quale!), quindi un assegnamento a `vet[3]` provoca una sovra-scrittura della memoria assegnata all'altra variabile, che molto probabilmente[14] modifica il valore della variabile stessa, con effetti imprevedibili sull'esecuzione del programma. Allo stesso modo, una lettura a tale locazione di memoria restituirà un numero intero che non ha nessun significato logico per il programma in esecuzione. In entrambi i casi, il comportamento del programma può divenire imprevedibile.

Indicizzare un vettore di `N` elementi, dichiarato per esempio come `int vett[N]`, nell'intervallo $[1 \ldots N]$, costituisce un errore semantico. Questo tipo di errore non viene segnalato dal compilatore.

6.5.2 Uso delle macro in combinazione con vettori e cicli

Un utile espediente per impostare i limiti dei cicli, utilizzato nell'esempio precedente, è quello di utilizzare le cosiddette *macro*. Una macro definisce del testo che viene sostituito nel codice sorgente prima della compilazione.

Un esempio banale è il seguente:

```
#define MAX (10)

int main()
{
  int vet[MAX];

  for (i = 0; i < MAX; i++)
    /* ... */
}
```

in ogni punto del programma viene usato `MAX` per fare riferimento alla dimensione del vettore, sia nella definizione del numero di elementi che dei limiti per l'iterazione su tali elementi (ed eventualmente anche in altri punti, in programmi più complessi). Il vantaggio è che cambiando il valore della macro e ri-compilando si adatta facilmente la dimensione del vettore a nuove esigenze, aggiornando automaticamente in un solo colpo tutti i punti del programma che fanno riferimento alla dimensione del vettore. Si consulti la Sezione 11.1 per maggiori dettagli sulla direttiva `#define`.

6.5.3 Le matrici

È possibile dichiarare anche array multi-dimensionali. Il caso tipico è quello di array a due dimensioni, ovvero le cosiddette matrici. Un esempio di dichiarazione di matrici è la seguente:

[14]Non è detto che ne modifichi il valore: potremmo essere talmente fortunati da scrivere in tale locazione un valore uguale a quello già presente, ma la probabilità che ciò accada è piuttosto bassa.

```
nome-tipo identificatore [ cardinalita' ] [ cardinalita' ] ;
```

dove

- *nome-tipo* è un qualsiasi tipo di dato, sia semplice che composto;
- *identificatore* è il nome che identifica la matrice;
- *cardinalità* è un numero intero che indica di quanti elementi è costituita la matrice; la prima e la seconda dimensione sono relative rispettivamente al numero di righe e di colonne. relativa

Si noti che una matrice può essere anche vista come un vettore i cui singoli elementi sono vettori essi stessi.

Un esempio di dichiarazione di matrice è il seguente:

```
int mat[40][5];
```

definisce una matrice denominata `mat` di 40 righe per 5 colonne. Le due componenti sono indicizzate da 0 a 39 e da 0 a 4 rispettivamente. Per esempio, è possibile riferirsi al valore `mat[5][1]`, il quale è un valore intero che può essere assegnato ed utilizzato nelle espressioni come una qualunque variabile di tipo intero.

Nell'esempio seguente viene calcolato il prodotto righe per colonne di due matrici `A` e `B` aventi dimensione opportuna, e il risultato viene memorizzato nella matrice `C`. Si ricorda che il prodotto matriciale $C = A \times B$, dove A ha dimensioni $m \times n$ e B ha dimensioni $n \times p$, genera una matrice C di dimensione $m \times p$ i cui elementi sono dati da

$$(C)_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \ldots + a_{in} b_{nj}$$

Nell'esempio si ha $m = 10$, $n = 3$ e $p = 20$. Si noti il fatto che la sommatoria al variare di r nell'equazione precedente avviene per r che varia da 1 ad n, mentre nel codice seguente r varia da 0 ad $n - 1$ per la nota necessità di indicizzare gli array entro tale intervallo.

Il programma completo è contenuto nel file `matrice_prodotto.c`.

```
1   int A[10][3], B[3][20], C[10][20];
2   int i, j, r;
3
4   for (i = 0; i < 10; i++) {
5     for (j = 0; j < 20; j++) {
6       C[i][j] = 0;
7       for (r = 0; r < 3; r++) {
8         C[i][j] = C[i][j] + A[i][r] * B[r][j];
9       }
10    }
11  }
```

Nell'esempio non è importante come i valori delle matrici siano stati assegnati alle matrici `A` e `B` da moltiplicare tra loro, ma importa notare come vengano gestiti gli indici per l'accesso agli elementi delle matrici. Si noti come l'utilizzo degli indici alla linea 8 sia consistente con la specifica della formula per il calcolo del prodotto tra matrici.

6.5.4 Array multi-dimensionali

È possibile definire array con un numero arbitrario di dimensioni. La sintassi per la dichiarazione di un array n-dimensionale è la seguente:

```
nome-tipo identificatore [ dim_1 ] [ dim_2 ] ... [ dim_n ] ;
```

Si noti che i vettori e le matrici sono casi particolari, e peraltro quelli più comuni, del precedente schema sintattico.

Per esempio, la dichiarazione:

```
double var[3][6][9][12];
```

dichiara un array a 4 dimensioni. Un elemento qualsiasi di questo array, per esempio `var[0][5][8][1]`, è un valore `double`. Ciascuna delle dimensioni può essere indicizzata da 0 a $m - 1$, dove m è il numero di elementi per quella specifica dimensione. Per esempio, l'indicizzazione `var[0][5][9][1]` per l'array dichiarato sopra non è corretta, in quanto gli indici validi per la terza dimensione dell'array devono variare nel range $[0 \ldots 8]$.

6.5.5 Memorizzazione degli array

Gli array multidimensionali vengono memorizzati per righe, in aree di memoria contigue. Dato un array n-dimensionale come il seguente:

```
float var [ dim_1 ] [ dim_2 ] ... [ dim_n ] ;
```

l'elemento

```
var [ i_1 ] [ i_2 ] ... [ i_n-1 ] [ i_n ];
```

si trova all'indirizzo che si ottiene dall'indirizzo del primo elemento, cioè l'*indirizzo* di

```
var [ 0 ] [ 0 ] ... [ 0 ]
```

al quale si somma

```
i_1 * dim_2 * ... * dim_n + ... + i_n-1 ... dim_n + i_n
```

6.6 STRUTTURE DATI

Una struttura dati è un tipo di dati composto, i componenti si chiamano *campi* e possono essere tipi semplici o altre strutture dati. Una struttura viene dichiarata nel seguente modo:

```
struct nome {
    tipo-campo nome-campo ;
   [tipo-campo nome-campo ;  ... ]
} ;
```

Dopo la dichiarazione, "struct nome" è il nome di un nuovo tipo che può essere usato per dichiarare variabili e puntatori. Esempio:

```
struct t_punto {
  int x;
  int y;
};

struct t_punto punto;      /* dichiara una variabile */
struct t_punto *punto_pt;  /* dichiara un puntatore */
```

dove si definisce una variabile di nome `punto` di tipo `struct t_punto`. Questo nome corrisponde ad una porzione di memoria in grado di conservare due dati (campi) di tipo `int` a cui si è dato il nome di `x` e `y`. Mentre la variabile `punto` può memorizzare due valori interi, la variabile `punto_pt` non può memorizzare altro che l'indirizzo di una struttura, poiché non è stata allocata memoria per la struttura ma soltanto per un puntatore ad essa.

Le strutture si possono inizializzare in tre modi diversi:

1. elencando i campi separati da virgola (sintassi tradizionale);
2. dichiarando i campi con i due-punti (sintassi di gcc fin da prima della standardizzazione);
3. usando l'assegnamento ai campi (sintassi standard C99, supportata anche dal gcc).

La prima forma è da evitarsi in quanto poco leggibile, la seconda è sconsigliata in quanto non standard. In tutti e tre i casi, ogni campo non esplicitamente inizializzato viene azzerato bit-per-bit dal compilatore. Nel seguente esempio le tre strutture sono uguali, con il campo `privato` inizializzato a zero:

```
struct elemento {
  int id;
  char *nome;
  int valore;
  int privato;
};

struct elemento i1 = {3, "aldo", 45};
struct elemento i2 = {id: 3, nome: "aldo", valore: 45};
struct elemento i3 = {.id = 3, .nome = "aldo", .valore = 45};
```

Un altro esempio di strutture dati è la seguente

```
struct data {
  int giorno;
  in mese;
  int anno;
} giorno1, giorno2;

struct data giorno3;
```

Si definiscono un nuovo tipo di nome `struct data` e contemporaneamente due variabili `giorno1` e `giorno2`. Poi una nuova variabile dello stesso tipo `giorno3`.

Per far riferimento ai singoli dati si usa la notazione

```
<nome variabile>.<nome campo>
```

Nell'esempio:

```
punto.x = 33;
punto.y = 44;

sqrt(punto.x * punto.x + punto.y * punto.y)
```

vengono prima assegnati dei valori ai campi della variabile `punto` di tipo `struct t_punto` definita predecentemente. Poi vengono utilizzati i valori contenuti nei campi per effettuare un calcolo matematico; in questo caso viene calcolata la norma del vettore identificato dal punto in questione, cioè la funzione

$$\sqrt{\text{punto.x}^2 + \text{punto.y}^2}$$

6.7 UNION

Oltre alle strutture, in C è possibile anche utilizzare le cosiddette union la cui definizione è molto simile a quella delle struct.

```
union nome {
    tipo-campo nome-campo ;
   [tipo-campo nome-campo ;  ... ]
} ;
```

Dove viene usata la parola chiave union invece di struct. Per esempio:

```
union union1 {
  short intero;
  char vet[2];
} un1, un2;
```

La differenza però è notevole: con le union si riserva spazio in memoria sufficiente solo per il dato più grande fra i vari campi descritti. In pratica si accede sempre alla stessa porzione di memoria, ma a seconda del campo che si utilizza per accedervi, i bit memorizzati sono interpretati in accordo con il tipo di dato specificato. Nell'esempio sopra riportato, la union occupa 2 byte, in quanto sia lo short che il vettore sono di dimensione 2 byte.

Per esempio:

```
un1.vet[0]='0'; un1.vet[1]='1';
printf("%x", un1.intero);
```

nella prima linea si accede alla memoria assegnando dei valori agli elementi del vettore, che in pratica costituiscono i 2 byte della variabile short. I due byte impostati separatamente, vengono poi stampati come intero short. Su alcuni sistemi viene stampato 3031 su altri 3130, a seconda che si tratti di macchine big-endian o little-endian.

NOTA	Generalmente un codice che contiene union non è portabile.

6.8 INTERI INDIPENDENTI DALLA PIATTAFORMA

Per rendere un programma indipendente dalla piattaforma, e quindi dalla dimensione del tipo intero, sono applicabili diverse soluzioni.

Il kernel Linux definisce (in <linux/types.h>) i seguenti tipi di dimensione e segno (unsigned o signed) noti:

```
u8    s8     u16    s16
u32   s32    u64    s64
```

Lo standard C99 definisce i seguenti tipi di dimensione e segno noto, il cui uso non è ancora molto diffuso:

```
uint8_t    int8_t    uint16_t   int16_t
uint32_t   int32_t   uint64_t   int64_t
intptr_t
```

L'ultimo tipo elencato è un intero della stessa dimensione di un puntatore (in pratica unsigned long).

La cosiddetta endianness è una proprietà della piattaforma di esecuzione (processore) che determina con quale ordine vengono memorizzati *i byte* di un numero formato da più byte. La differenza consiste nell'ordine con il quale i byte sono memorizzati:

- big-endian è la memorizzazione che inizia dal byte più significativo per finire col meno significativo
- little-endian è la memorizzazione che inizia dal byte meno significativo per finire col più significativo

Per esempio, il valore esadecimale a 32 bit 0x01234567 (0x01 è il byte più significativo) viene memorizzato come 0x67|0x45|0x23|0x01 su macchine little-endian e 0x01|0x23|0x45|0x67 su macchine big-endian.
La differenza non riguarda la posizione dei bit all'interno del byte – in questo caso si parla di ordine dei bit – né la posizione dei caratteri in una stringa.

Approfondimento 1: Memorizzazione big-endian e little-endian.

6.9 CONVERSIONI DI TIPO

Le conversioni di tipo, o *casting*, permettono di trasformare il valore contenuto in una variabile di un determinato tipo in un valore opportuno di un tipo diverso. La conversione può avvenire in vari modi.

Le conversioni *automatiche* prevedono che, in espressioni che coinvolgono tipi diversi, il risultato dipende dall'operando più ricco di informazioni. Per esempio:

- 8/5 coinvolge due interi, il risultato è l'intero di valore 1;
- 8/5.0 coinvolge un intero ed un double, il risultato è double 1.6;

Vi sono conversioni *per assegnamento*, nelle quali il valore assegnato viene convertito nel tipo della dell'espressione a sinistra dell'operatore di assegnamento. Esempi di tale conversione sono:

1. `float` → `int` vengono troncati i decimali
   - es. `int n;` `n = 1.6;` `n` vale 1
   - es. `int n;` `n = -1.6;` `n` vale -1 (non vi sono arrotondamenti)
2. `int` → `char` vengono conservati i bit meno significativi
   - es. `char c;` `c = 257;` `c` vale 1

Infine, esistono le conversioni *esplicite* (operatore cast), nella forma

```
(nome tipo) espressione
```

nelle quali si forza esplicitamente la conversione nel tipo di dato specificato. Per esempio:

- il risultato dell'espressione `8/(double)5` è 1.6 (il valore 5 viene forzato a `double`)
- il risultato dell'espressione `(double)(8/5)` è il valore `double` 1.0 (prima si calcola la divisione fra interi, poi il risultato viene convertito in `double`)

6.10 ASSEGNARE NUOVI NOMI AI TIPI DI DATO: TYPEDEF

In C è possibile assegnare dei nomi simbolici ai tipi di dati esistenti. Questo viene fatto principalmente per migliorare la chiarezza di programmi lunghi e complessi, oppure per costituire dei tipi di dati indipendenti dalla piattaforma sulla quale il programma sarà eseguito per motivi di portabilità. Per esempio, si può pensare di fare in modo che un tipo intero abbia sempre dimensione di 16 bit. Verrà perciò definito un nome simbolico del tipo

```
int16
```

che verrà ridefinito per le diverse architetture a seconda della dimensione dei tipi base. Perciò, `int16` potrà corrispondere a un `int` su certe architetture, oppure a uno `short` su altre. Ma un qualsiasi programma applicativo userà sempre il nome `int16`.

La definizione dei nuovi tipi si realizza per mezzo della parola chiave `typedef`, con la sintassi

```
typedef tipo nome;
```

che associa il nome `nome` al tipo `tipo`. Nell'esempio precedente si scriverebbe

```
typedef int int16;
```

In un altro esempio, un sistema di calcolo che tiene traccia del trascorrere del tempo in unità discrete potrebbe effettuare la seguente definizione:

```
typedef unsigned long TIME;
```

Questo permette, all'interno del programma, di individuare facilmente le variabili che hanno a che fare col tempo, in quanto sono "dichiarate tipo `TIME`", distinguendole da generiche variabili di tipo `unsigned long` utilizzati per altri scopi.

Il fatto di affermare che le variabili sono "dichiarate di tipo `TIME`" è un po' impropria. Infatti, l'assegnazione del nome `TIME` al tipo `unsigned long` *non crea un nuovo tipo di dato*: dal punto di vista semantico una variabile dichiarata di tipo `unsigned long` è perfettamente equivalente ad una di tipo `TIME`. L'assegnazione di nuovi nomi ai tipi di dato potrebbe essere realizzata in modo simile utilizzando la direttiva del preprocessore `#define`, con la differenza che nel caso di `typedef` la sostituzione viene fatta dal compilatore e non dal preprocessore, che garantisce la coerenza di controllo sulla tipizzazione delle varibili. Infatti, la direttiva `#define` serve per effettuare delle sostituzioni tipografiche nel testo. Questa direttiva viene descritta in dettaglio nella Sezione 11.1.

In altri casi è possibile assegnare un nome sintetico a tipi complessi, in modo da aumentare la chiarezza del codice. Nell'esempio seguente viene associato il nome `t_cerchio` alla struttura che contiene i dati per rappresentare un cerchio:

```
typedef struct {
  int x,
  int y;
  int raggio;
} t_cerchio;
```

in questo modo si possono definire e utilizzare variabili di tipo `t_cerchio`, per esempio definendo una funzione che controlla se due cerchi sono uguali nel modo seguente:

```
int uguale(t_cerchio c1, t_cerchio c2)
{
  return ((c1.x == c2.x) && (c1.y == c2.y) && (c1.raggio == c2.raggio));
}
```

6.10.1 `typedef` vs. `#define` per definite nuovi tipi

In precedenza (Sezione 6.5.2) è stato illustrato l'utilizzo di macro per definire delle etichette che vengono sostituite all'interno del codice sorgente dal pre-processore prima che avvenga la compilazione vera e propria. Maggiori informazioni sulla direttiva `#define` sono fornite nel Capitolo 11.1. Questo potrebbe suggerire l'utilizzo delle macro create con `#define` per gestire dei tipi di dati. Questa è effettivamente una possibilità concreta, ma che presenta alcuni inconvenienti che ne rendono sconsigliabile l'utilizzo.

Il programma di esempio è contenuto nel file `typedef_define.c`.

Le seguenti istruzioni definiscono delle nuove etichette per il tipo `char` e puntatore a `char` (`char *`), utilizzando sia le `#define` che la `typedef`:

```
1  #include <stdio.h>
2
3  #define T_CHAR_D char
4  #define T_CHARP_D char *
5  typedef char t_char_t;
6  typedef char * t_charp_t;
7
8  typedef char t_char_vect[];
9
10 int main()
11 {
12   t_char_vect v = "la mia stringa\n";
13
14   /* due dichiarazioni equivalenti */
15   t_char_t n2;
16   T_CHAR_D n1;
17
18   //unsigned t_char_t n3;
19   unsigned T_CHAR_D n4;
20
21   t_charp_t p1, p2, p3;
22   T_CHARP_D p4, p5, p6;
23
24   printf("%ld %ld %ld\n", sizeof(p1), sizeof(p2), sizeof(p3));
25   printf("%ld %ld %ld\n", sizeof(p4), sizeof(p5), sizeof(p6));
26
27   return 0;
28 }
```

I nomi delle etichette alle line [3. . .6] sono esplicativi, e si interpretano in questo modo: la 't' iniziale sta per *tipo*, segue il tipo base, che è `char`, e una 'p' è presente per indicare il puntatore. Infine la 'D' o la 't' finale discrimina tra un'etichetta realizzata rispettivamente con la `#define` che la `typedef`. Per convenzione, le `#define` sono universalmente scritte in caratteri maiuscoli, mentre per le `typedef` la convenzione non è così stretta; la convenzione adottata in questo testo è di scriverle in minuscole.

Date le istruzioni precedenti, si ha che le seguenti due istruzioni alle linee 16 e 15 sono perfettamente equivalenti nel dichiarare due variabili di tipo `char`. La differenza consiste nel fatto che il compilatore è a conoscenza dell'esistenza dell'etichetta `t_char_t`, mentre la macro viene sostituita dal pre-processore prima della compilazione, e in questo modo il compilatore trova la stringa

```
char n1;
```

all'interno del file sorgente.

Un primo aspetto di differenza tra i due approcci è il seguente, nel quale si intende utilizzare `unsigned` per specificare che la variabile dichiarata è senza segno:

```
unsigned t_char_t n3;  /* questo non si puo' fare!!! */
unsigned T_CHAR_D n4;
```

L'istruzione 18 genera un errore di compilazione, e quindi non è permessa, mentre l'istruzione 19 è lecita, in quanto viene vista dal compilatore come

```
unsigned char n4;
```

dopo che il testo `T_CHAR_D` è stato sostituito con `char` dal pre-processore.

Se si ritiene erroneamente di poter utilizzare una macro con le stesse finalità della `typedef`, si possono commettere errori sottili come nel caso delle linee 21 e 22. Nella prima vengono dichiarati 3 puntatori a carattere, mentre la seconda istruzione dichiara un puntatore a carattere (il primo) e due variabili di tipo `char`. Infatti, sostituendo tipograficamente all'etichetta `T_CHARP_D` il valore specificato nella macro, si ottiene

```
char * p4, p5, p6;
```

Dal momento che l'asterisco fa parte del dichiaratore (cioè è associato all'identificatore) e non del tipo di dato, per dichiarare tre puntatori a carattere si sarebbe dovuto scrivere

```
char *p4, *p5, *p6;
```

cosa evidentemente impossibile da realizzare con una macro come quella definita, a meno di non scrivere

```
T_CHARP_D p4;
T_CHARP_D p5;
T_CHARP_D p6;
```

che però fa venire meno la comodità di dichiarare le tre variabili con una sola linea di codice.

Infine, la `typedef` permette di definire dei vettori, cosa che non è fattibile utilizzando la macro. Per esempio, il seguente codice è corretto:

```
typedef char t_char_vect[];

t_char_vect v = "la mia stringa\n";
```

ed equivale alla dichiarazione

```
char v[] = "la mia stringa\n";
```

In conclusione, è sconsigliabile l'uso delle macro per la definizione di nuovi tipi di dato. Per questa finalità è meglio sfruttare le possibilità offerte dalla `typedef`.

Capitolo 7

I PUNTATORI

La memorizzazione di una variabile richiede un certo numero di byte. La variabile è collocata ad un particolare indirizzo di memoria. Un puntatore è un tipo di variabile che serve a memorizzare *un indirizzo di memoria*. È possibile, per esempio, memorizzare l'indirizzo di un'altra variabile del programma.

La dimensione di un puntatore, cioè la quantità di memoria allocata per la memorizzazione di un indirizzo, è sempre uguale a quella di un intero lungo, cioè di un `long`.

In Figura 7.1 è rappresentata la variabile `num` di dimensione 4 byte, il cui indirizzo è pari a 104. Un puntatore che punta alla variabile `num` conterrà quindi il valore 104. La situazione è riassunta in Tabella 7.1. Si noti che l'indirizzo *a cui è allocato* il puntatore `ptr`, che è pari a 116, non è rilevante in questo caso, in quanto potrebbe essere allocato indifferentemente in qualsiasi punto della memoria.

I puntatori sono un tipo di dato elementare usato per accedere alla memoria e manipolare indirizzi. Una variabile di tipo puntatore permette di memorizzare l'indirizzo di una variabile.

La dichiarazione:

```
int *ptr, num;
```

dichiara una nuova variabile `ptr` di tipo puntatore, dove l'oggetto puntato è di tipo intero.

L'operatore unario & permette di ottenere l'indirizzo di una variabile. Per esempio, l'istruzione

```
ptr = &num;
```

in `ptr` è stato memorizzato l'indirizzo della variabile `num`. Quindi se `num` è memorizzata all'indirizzo 104, dopo l'assegnazione il valore di `ptr` varrà 104.

Analogamente l'operatore unario * permette di ottenere il valore contenuto in un particolare indirizzo di memoria

```
int val;
```

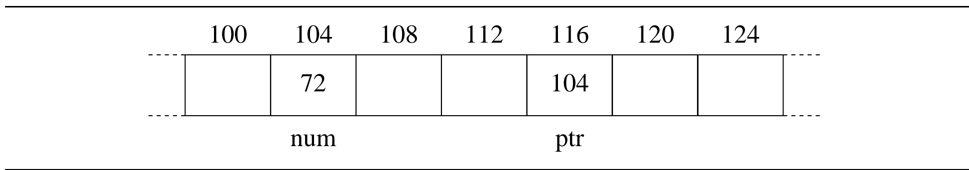


Figura 7.1 Una variabile in memoria; è rappresentato l'intervallo di indirizzi assoluti da 100 a 124.

Tabella 7.1 Lo stato rappresentato in Figura 7.1.

variabile	indirizzo	valore
num	104	72
ptr	116	104

```
val = *ptr;
```

Con i valori dell'esempio, `ptr` punta all'indirizzo della variabile `num`, la quale contiene il valore 72. Quindi `*ptr` indica il valore intero memorizzato all'indirizzo puntato da `ptr`, per cui a `val` sarà assegnato il valore 72.

Per ottenere il valore memorizzato devono essere fatti 2 accessi in memoria: il primo accesso viene fatto all'indirizzo di `ptr`, per recuperare il dato ivi memorizzato, ovvero l'indirizzo di `num`. Il secondo accesso avviene all'indirizzo di `num`, per recuperare il valore memorizzato nella variabile `num`.

7.1 PUNTATORI A `VOID`

La parola chiave `void` può essere utilizzata per dichiarare dei puntatori che non puntano a nessun tipo di dato in particolare. La seguente dichiarazione:

```
void *ptr;
```

dichiara appunto un puntatore a `void`. La dichiarazione di puntatori a `void` è molto utile nella scrittura di funzioni generiche che debbano operare su puntatori a tipi generici, per esempio poiché non si conoscono a priori. Il puntatore verrà quindi dichiarato `void *`.

Da notare che *non è possibile dichiarare variabili* `void`, cioè che non siano puntatori a `void`, infatti la seguente dichiarazione produce un errore in compilazione:

```
void variabile;
```

7.2 PUNTATORI E VETTORI

Per definizione, il nome di una variabile di tipo vettore rappresenta l'indirizzo costante dell'elemento di indice 0 del vettore stesso. Sempre per definizione, se `p` è un puntatore ad un particolare elemento di un vettore, `p+1` punta all'elemento successivo, `p+i` punta all'elemento che segue dopo `i` posizioni.

In Figura 7.2 è rappresentato del vettore `vet` di 4 elementi interi, di 4 byte ciascuno, allocato a partire dall'indirizzo 108 in memoria. Si può notare come il primo elemento del vettore, quello di indice 0, è indicato semplicemente con `vet`, mentre l'i-esimo elemento è indirizzabile con il costrutto sintattico `vet + i`.

Il programma, in una versione leggermente estesa, è contenuto nel file `vett_ptr.c`.

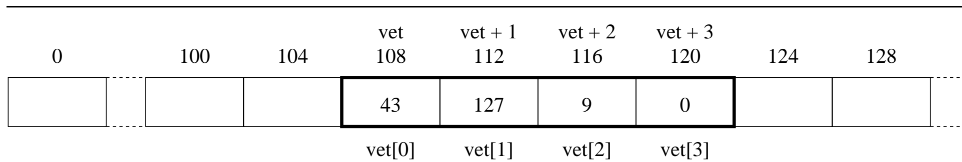


Figura 7.2 Vettore allocato in memoria e aritmetica dei puntatori utilizzata per accedere ai singoli elementi; i valori `vet`, `vet+1`, ecc. rappresentano indirizzi relativi all'indirizzo base del vettore che corrisponde all'indirizzo `vet`.

Nel seguente esempio:

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int vett[10];
6    int *p;
7    int i;
8
9    p = vett;   /* equivalente a p = &vet[0] */
10
11   for (i = 0; i < 10; i++)
12     *(p + i) = 100 + i;
13
14   for (i = 0; i < 10; i++)
15     printf("%d ", vett[i]);
16   printf("\n");
17
18   return 0;
19 }
```

vengono dichiarati il vettore `vett` di 10 elementi interi (linea 5), e il puntatore ad intero `p` (linea 6). L'istruzione 9 assegna al puntatore `p` l'indirizzo del primo elemento del vettore `vett`. Il vettore viene riempito con di linea 11, accedendo ai singoli elementi attraverso il puntatore `p`. Per mostrare che effettivamente gli elementi del vettore sono stati correttamente scritti, tali elementi vengono scritti a video indicizzando i singoli elementi mediante l'operatore "parentesi-quadre" alla linea 15.

Il C è pensato per essere molto vicino al processore e alle sue strutture; non dovrebbe perciò stupire la scelta degli autori del linguaggio di rappresentare un vettore con l'indirizzo del suo primo elemento. Puntatori e vettori sono perciò concetti intercambiabili e ad un puntatore si può applicare un indice tramite l'operatore parentesi-quadre senza che questo provochi errori o messaggi di attenzione. Il nome di un vettore è diverso da un puntatore in quanto non gli può essere assegnato un nuovo valore: si tratta di un indirizzo costante istanziato all'atto della compilazione del programma. Date le dichiarazioni dell'esempio sopra, la seguente istruzione produrrebbe un errore in compilazione:

```
vett = p;
```

in quanto si tenta di assegnare un valore a `vett`, che è un indirizzo costante.

I concetti di puntatore e vettore sono quindi molto vicini tra loro, anche se sono comunque due cose ben diverse. Un vettore è infatti un insieme di elementi omogenei allocati in un'area contigua di memoria. Un puntatore è invece una variabile che contiene un generico indirizzo in memoria.

Il modo corretto di accostare i concetti di vettore e puntatore è di considerare che *l'aritmetica dei puntatori è equivalente al metodo di indicizzazione dei vettori*.

Ai puntatori possono essere sommati e sottratti numeri interi: il risultato della somma di un puntatore e di un numero intero `n` è il puntatore all'elemento numero `n` del vettore. Il numero intero non rappresenta cioè il *numero di byte* da aggiungere nell'indirizzo, ma il *numero di elementi*; il fattore di scala appropriato viene applicato dal compilatore in base al tipo cui punta il puntatore oggetto dell'operazione aritmetica. È possibile quindi incrementare/decrementare un puntatore, come pure fare la differenza (ma non la somma) tra puntatori dello stesso tipo, e il risultato in questo secondo caso è un numero intero. L'aritmetica su puntatori generici (puntatori a `void`) non è definita secondo lo standard del linguaggio, mentre con `gcc` usa 1 byte come dimensione dell'elemento puntato.

Il "puntatore nullo" vale zero e non è un puntatore valido. Tipicamente le funzioni che ritornano un puntatore restituiscono un puntatore "nullo" come segnalazione di errore. La macro NULL vale 0, e 0 è confrontabile con qualunque puntatore. Per esempio è possibile scrivere un blocco di codice del tipo

```
char *ptr;
if (ptr == NULL) {
  /* istruzioni */
}
```

il quale esegue le istruzioni specificate solo se `ptr` è un puntatore nullo.

Gli operatori più importanti per usare i puntatori sono "*" (si legge "il puntato da") e "&" ("l'indirizzo di").

Esempi:

```
int i, v[10], *p; /* un numero intero, un vettore e un puntatore:
                     "e' intero ogni elemento del vettore v, di dimensione 10"
                     "e' intero l'oggetto puntato da p" */
p = v;          /* assegno a p il valore di v,
                   ossia l'indirizzo del primo elemento */
p = &v[0];      /* come sopra */
p = &v[4];      /* assegno a p l'indirizzo del quinto elemento di v */
p = v + 4;      /* come sopra */
p++;            /* incremento l'indirizzo p di 1
                   se p vale v+4, l'istruzione equivale a p = v + 5 = &v[5] */
i = p - v;      /* assegno 5 a i (in conseguenza delle due righe precedenti) */
```

Nell'esempio che segue, viene effettuato un ciclo fintanto che non viene incontrato un valore nullo nel vettore `v`. Il puntatore `p` viene utilizzato per scorrere il vettore, essendo inizializzato all'indirizzo del primo elemento del vettore stesso. Il ciclo termina quando il valore puntato da `p`, cioè `*p`, è nullo (si sicordi che il valore 0 equivale alla condizione logica "falso"). Il ciclo calcola la somma di tutti i valori considerati, memorizzandola in `sum`. Si noti che deve esserci almeno un elemento di `v` che vale zero, altrimenti il puntatore assumerà valori non validi andando ad accedere aree di memoria poste oltre la fine del vettore. Il compilatore non effettua alcun controllo implicito su puntatori e indici di vettori, quindi questo è un tipico errore logico dalle conseguenze spesso imprevedibili. Infatti tali conseguenze dipendono dal contenuto della memoria posta dopo quella allocata per il vettore `v`.

```
sum = 0;
for (p = v; *p; p++)
  sum += *p;
```

Fare assegnamenti tra puntatori di tipo diverso è una operazione che viene segnalata con un messaggio di avvertimento, a meno di non effettuare un casting esplicito. È comunque possibile convertire un puntatore da un tipo ad un'altro, ma anche un puntatore in numero intero e viceversa; queste

conversioni non provocano la generazione di codice macchina, perché comunque nel processore i puntatori sono rappresentati da numeri interi, ma sono necessarie per la pulizia semantica del codice sorgente.

È sempre consentito l'assegnamento di un puntatore-a-void a qualunque altro tipo di puntatore, come pure l'assegnamento di qualunque puntatore ad un puntatore-a-void. Questo perché il tipo "`void *`" è quello che normalmente si usa per gestire indirizzi generici di memoria, operazione comunissima nei sistemi operativi e nelle librerie di sistema.

Se sono state utilizzate le seguenti dichiarazioni e istruzioni:

```
int vet[100];
int *p, *q;
int x, y, z, i;

p = &vet[0];   /* equivalente a p = vet */
```

le seguenti istruzioni sono equivalenti:

```
x = vet[0];      x = *p;        x = *vet;
y = vet[i];      y = *(p+i);    y = *(vet + i);
vet[i] = z;      *(p+i) = z;    *(vet + i) = z;
q = &vet[0];     q = p;         q = vet;
```

7.3 L'OPERATORE `SIZEOF`

L'operatore `sizeof`, applicato ad un tipo, ad un nome di variabile o ad un'espressione, ritorna la dimensione in byte dell'oggetto indicato. Tale calcolo viene effettuato in compilazione in base al tipo di dato che viene passato a `sizeof`.

Se si incrementa un puntatore `p`, il suo valore numerico (indirizzo in memoria in byte) viene incrementato di `sizeof(*p)`, ossia della dimensione dell'oggetto puntato.

Alcuni esempi:

```
int i, v[10], *p;          /* le stesse variabili di prima */

i = sizeof(int);           /* normalmente 4, ma puo' essere 8, oppure 2 */
i = sizeof(i);             /* come sopra */
i = sizeof(v[0]);         /* come sopra */
i = sizeof(*p);            /* come sopra */
i = sizeof(p);            /* 4 (dimensione del puntatore), oppure 8 */
i = sizeof(v);             /* 40, oppure 80, oppure 20 */
i = sizeof(v)/sizeof(*v); /* 10: il numero di elementi nel vettore v */

#define ARRAY_SIZE(x) (sizeof(x)/sizeof(*x)) /* una comoda macro */
```

La macro si può utilizzare passandole come argomento un qualsiasi oggetto, per esempio l'identificatore di un vettore:

```
int dim, v[10];

dim = ARRAY_SIZE(v);   /* dim conterra' il valore 10 */
```

Essendo un operatore, `sizeof` può essere utilizzato ponendo l'operando tra parentesi oppure anche senza l'utilizzo delle parentesi. Per esempio, le seguenti istruzioni sono equivalenti tra loro:

```
int a[100], b[50];
```

```
int size_a, size_b;

/* se sizeof(int) = 4, allora size_a vale 400 */
size_a = sizeof a;
size_a = sizeof(a);

/* se sizeof(int) = 4, allora size_b vale 200 */
size_b = sizeof b;
size_b = sizeof(b);
```

e si noti appunto la possibilità di specificare l'operando tra parentesi o senza l'utilizzo delle parentesi.

Un'eccezione a questa possibilità si ha nel caso seguente:

```
int size_int;

size_int = sizeof(int);   /* espressione valida */
size_int = sizeof int;    /* errore in compilazione */
```

Quando l'operando di `sizeof` è il nome di un tipo di dato (`int`, nell'esempio), le parentesi sono obbligatorie. Questo permette di evitare ambiguità nel parsing dell'espressione[15].

7.4 LE STRINGHE

Una stringa costante è una sequenza di 0 o più caratteri racchiusi fra doppi apici. Per esempio:

```
"Questa e` una stringa"

"Le due stringhe in fase di compilazione"
" saranno concatenate"
```

La stringa vuota è "".

Una variabile stringa viene definita come segue:

```
char stringa[10];
```

è cioè, se n è la dimensione del vettore, si tratta di un vettore di caratteri in grado di memorizzare al più $n - 1$ caratteri; l'ultimo carattere di una stringa è sempre zero.

NOTA Il carattere di valore ASCII 0 (zero) si può indicare indifferentemente con il valore intero 0 oppure con il carattere '

0'. Da notare che esso è diverso dal carattere '0', il cui valore intero corrispondente è 48.

[15] In questo caso, per parsing si intende il riconoscimento degli elementi che compongono l'espressione da parte del compilatore.

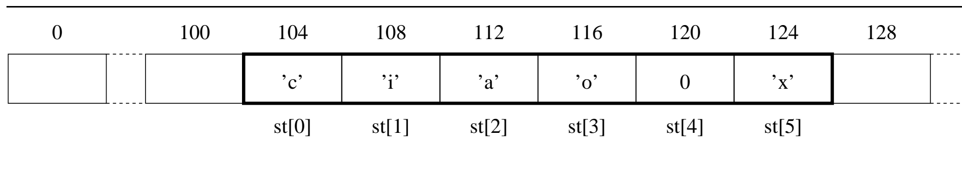


Figura 7.3 Vettore di 6 caratteri contenente la stringa `"ciao"`.

Quando si parla di variabili stringa, dichiarate come vettori di caratteri, si deve tenere presente che la lunghezza della stringa, ovvero il numero di caratteri significativi della stringa stessa, può essere ben diversa dalla dimensione del vettore: non tutti i caratteri del vettore sono sono necessariamente usati dalla stringa. In questo caso, il carattere 0 indica dove termina la stringa.

La Figura 7.3 mostra il vettore `st` di 6 caratteri, cioè dichiarato come

```
char st[6];
```

Il vettore è memorizzato all'indirizzo 104, e contiene la stringa `"ciao"` di 4 caratteri (si osservi il contenuto dei primi 4 caratteri del vettore), che viene terminata dal carattere 0.

La stringa

```
char s[] = "abcde";
```

è una stringa inizializzata. In modo equivalente si poteva scrivere

```
char s[] = { 'a', 'b', 'c', 'd', 'e', '\0' };
```

Di fatto, una stringa in C è un vettore di caratteri terminato da un byte a 0. Non possono quindi contenere il carattere '\0'. La rappresentazione tra virgolette è solo una notazione semplificata per rappresentare un vettore. Ogni volta che nel testo del programma appare una stringa tra virgolette, il compilatore memorizza la stringa nel segmento dati del programma e la rappresenta con l'indirizzo del suo primo elemento. Un carattere incluso in apici singoli è un numero intero, cioè il codice ASCII del carattere indicato.

Riassumendo, esempi di dichiarazioni di stringhe e puntatori:

```
/* un vettore di 6 caratteri, compreso il terminatore */
char s[] = "prova";

/* lo stesso, in notazione barocca */
char s[] = {'p', 'r', 'o', 'v', 'a', 0};

/* un carattere, un puntatore a carattere */
char c, *t;

/* c prende il valore 'p' */
c = *s;

/* t rappresenta la stringa "ova" */
t = s + 2;

/* ora la stringa s e' "trova" */
s[0] = 't';

/* un puntatore ad un'area preiinizializzata di 7 byte */
```

```
char *name = "arturo";

/* un'area di 6 byte, con indirizzo "surname" */
char surname[] = "rossi";

/* ora name indica "rturo" */
name++;

/* errore: surname \'e un indirizzo costante */
surname++;
```

7.4.1 Dettagli sull'inizializzazione

Si è detto che vettori e puntatori sono concetti molto vicini tra loro, in particolare l'aritmetica dei puntatori è identica al metodo di indicizzazione dei vettori.

Ci sono però delle sottili differenze che è bene tenere in considerazione quando si utilizzano vettori o puntatori credendo che siano completamente intercambiabili.

Le seguenti due istruzioni possono essere utilizzate per usare la stringa relativa:

```
char *nome = "paolo";
char nome[] = "paolo";
```

Le due istruzioni dichiarano rispettivamente un puntatore e un vettore. Di conseguenza, lo spazio occupato in memoria dovuto alla prima istruzione è pari a quello di un puntatore più lo spazio necessario a memorizzare la stringa. Nel secondo caso, lo spazio occupato è pari alle dimensioni del vettore. Come per tutti i puntatori, è possibile cambiare il valore dell'indirizzo memorizzato nel corso del programma, mentre nel caso del vettore, il nome è un indirizzo costante che non può essere modificato.

La stringa memorizzata nel vettore può essere modificata in qualsiasi momento, accedendo ai singoli elementi del vettore oppure utilizzando funzioni come `strcpy`[16]. Per quanto riguarda il puntatore, la relativa stringa viene memorizzata come stringa *senza nome* in un'area della memoria scelta dal compilatore. Quest'area di memoria può anche essere a *sola lettura*, quindi non è detto che il valore contenuto nella stringa risulti modificabile, portando ad errori in esecuzione del programma se si tenta una modifica.

Inoltre, se è ammesso effettuare degli assegnamenti come quelli fatti sopra in fase di dichiarazione di una stringa, non è possibile assegnare una stringa mediante l'usuale operatore di assegnamento = se non al momento della sua dichiarazione. Il codice seguente, infatti, genera un errore al momento della compilazione:

```
char nome[10];
nome = "paolo";
```

Le stringhe, come i vettori, non possono essere assegnate come un "tutt'uno", ma è necessario assegnare ciascun carattere esplicitamente, oppure usare la funzione `strcpy` come segue:

```
char nome[10];
strcpy(nome, "paolo");
```

La funzione copia la stringa `"paolo"` all'interno del vettore `nome`. La funzione sostanzialmente copia ogni carattere del secondo parametro a partire dall'indirizzo specificato dal primo parametro, fino a quando non incontra un carattere nullo. Il primo parametro deve essere un puntatore a carattere: tutto

[16]Sta per [str]ing [c]o[py].

funziona correttamente poiché, come sappiamo, il nome di un vettore (di caratteri, in questo caso), corrisponde all'indirizzo del suo primo carattere.

Se si volesse inserire la stringa `"paolo"` all'interno del vettore `nome` a partire dal secondo carattere, si potrebbe utilizzare indifferentemente una delle due seguenti istruzioni:

```
strcpy(&nome[1], "paolo");
strcpy(nome + 1, "paolo");
```

Ovviamente ci si deve assicurare che il primo carattere del vettore `nome` non sia nullo, altrimenti per quanto abbiamo scritto i caratteri correttamente a partire dal secondo elemento del vettore, `nome` apparirà come una stringa vuota (il primo carattere è nullo!).

7.4.2 Esempio di funzione per la manipolazione delle stringhe

Alla luce di quanto appreso riguardo alle stringhe, un modo per determinare la lunghezza di una stringa può quindi essere il seguente[17]:

```
int lunghezza;
char vet[10];

strpcy(vet, "prova");

char *t = vet;
for (; *t; t++)
   ;
lunghezza = t - vet;
```

La lunghezza della stringa viene calcolata come differenza tra l'indirizzo del carattere '\0' (che verrà puntato da `t` al termine del ciclo) e l'indirizzo del primo carattere della stringa, corrispondente al valore `vet`. Il ciclo infatti termina quanto `*t` è nullo, dal momento che per il C la condizione logica falsa è associata al valore intero 0.

La funzione `strlen` è già disponibile nella librerie di sistema, includendo il file di intestazione <`string.h`>, e si utilizza nel seguente modo:

```
lunghezza = strlen(vet);
```

Non è quindi normalmente necessario doverla riscrivere.

Da notare che il corpo del ciclo `for` non contiene alcuna istruzione: il ciclo viene effettuato con il solo scopo di incrementare il valore dell'indirizzo contenuto nella variabile locale `t`. Dal momento che il fatto di mettere un punto e virgola subito dopo un'istruzione `for` è un errore abbastanza comune, per rendere palese che il ciclo non contiene istruzioni, si mette il punto e virgola a capo. Questo fa intendere a chi legge questo spezzone di codice che *è corretto* che il ciclo non contenga istruzioni. In alternativa, si sarebbe anche potuto scrivere:

```
  for (; *t; t++) { }
```

7.5 VETTORI DI PUNTATORI/STRINGHE

Dal momento che il tipo elementare che forma un vettore può essere uno qualsiasi dei tipi base del linguaggio, possono essere dichiarati anche vettori di puntatori e, per estensione, vettori di stringhe. Per esempio, l'istruzione seguente dichiara un vettore di 5 puntatori a carattere:

[17] La funzione di libreria `srtlen` funziona in questo modo identico a questo.

```
char *nomi[5];
```

Come normalmente avviene per i vettori, i 5 puntatori vengono allocati in aree contigue della memoria, e possono essere utilizzati per puntare a delle stringhe. Con la dichiarazione precedente, non viene allocato spazio per le stringhe di caratteri, ma solo per dei puntatori a stringhe che dovranno essere allocate successivamente.

I vettori di stringhe possono essere inizializzati come qualsiasi altro vettore al momento della dichiarazione, assegnando per esempio una stringa ad ogni elemento del vettore. Per esempio:

```
char *nomi[5] = {"Sara", "Paolo", "Giovanni", "Elvira", "Sebastiano"};
```

La precedente istruzione è equivalente alla seguente:

```
char *nomi[] = {"Sara", "Paolo", "Giovanni", "Elvira", "Sebastiano"};
```

nella quale però il vettore viene dimensionato automaticamente in base al numero di elementi specificati nell'inizializzazione.

Ciascuna stringa viene messa dal compilatore in una locazione non nota a priori, e il puntatore al primo elemento della stringa viene memorizzato nel vettore.

7.6 PUNTATORI E STRUTTURE

È molto comune effettuare delle dichiarazioni di puntatori a strutture. Nell'esempio seguente

```
struct t_punto {
  int x;
  int y;
};
struct t_punto *ptr;
```

viene dichiarato un puntatore `ptr` che punta ad una struttura di tipo `struct t_punto`. Come detto più volte, in questo modo non viene allocata memoria per una struttura, ma soltanto per un puntatore. Con le seguenti istruzioni:

```
struct t_punto punto;
ptr = &punto;
```

viene invece allocata memoria per la variabile `punto` di tipo `struct t_punto`, e il puntatore `ptr` viene fatto puntare a tale variable con il successivo assegnamento.

È possibile accedere ora ai singoli campi della variabile `punto` utilizzando l'operatore *punto*, per esempio assegnando dei valori ai campi:

```
punto.x = 10;
punto.y = 40;
```

oppure

```
sqrt(punto.x * punto.x + punto.y * punto.y);
```

Quando si utilizzano i puntatori per accedere ai campi delle strutture, è necessario adoperare l'operatore "freccia", composta dal trattino e dal maggiore ($->$), come nell'esempio seguente:

```
ptr->x = 20;
ptr->y = 50;
```

oppure

```
sqrt(ptr->x * ptr->x + ptr->y * ptr->y);
```

nel quale vengono assegnati due nuovi valori ai campi, accedendovi per mezzo del puntatore `ptr`. Chiaramente tale accesso è possibile soltanto in quanto precedentemente è stato assegnato a `ptr` l'indirizzo della struttura `punto`.

7.6.1 Puntatori a strutture come campi di strutture

Una struttura dati può includerne altre o includere puntatori ad altre strutture. Utilizzando la struttura `struct t_punto` dichiarata precedentemente, è possibile includerla in una struttura più complessa come nell'esempio seguente:

```
struct t_rettangolo {
  struct t_punto p1;
  struct t_punto p2;
};
```

Per accedere ai valori delle variabili memorizzate nei campi si utilizza sempre l'operatore punto:

```
struct t_rettangolo rett;
rett.p1.x = 10;
rett.p1.y = 20;
rett.p2.x = 30;
rett.p2.y = 40;
```

Se si interpretano i due punti che definiscono il rettangolo come, rispettivamente, l'angolo in alto a sinistra e quello in basso a destra, è possibile utilizzare le informazioni memorizzate nella struttura per calcolare l'area del rettangolo come segue:

```
int area;
area = abs((rett.p1.x - rett.p2.x) * (rett.p1.y - rett.p2.y));
```

dove `abs` è la funzione che restituisce il valore assoluto del suo argomento.

Mentre i puntatori possono riferire una struttura da un'altra ciclicamente, l'inclusione di strutture non può essere ricorsiva, perché la struttura inclusa è interamente contenuta nella struttura includente. Poiché il compilatore effettua una sola passata sul codice, se una struttura contiene il puntatore ad un'altra struttura, occorre dichiarare tale struttura preventivamente, anche senza definirne l'elenco dei campi. Tale struttura non può essere istanziata, perché il compilatore non sa la sua dimensione in byte, ma si possono definire puntatori ad essa, perché i puntatori hanno tutti la stessa dimensione.

Esempio:

```
struct padre;

struct figlio {
  struct padre *padre;
  /* ... */
};

struct padre {
  struct figlio *figlio;
  /* ... */
};
```

Dichiarare una struttura senza specificarne l'elenco dei campi permette anche di creare strutture *opache* in una libreria, normalmente usate per dati privati della libreria stessa. Se una struttura contiene un puntatore alla struttura stessa non serve la dichiarazione preventiva, perché mentre il compilatore legge la lista dei campi ha già visto il nome della struttura stessa.

```
struct dati_privati;
struct dato {
  struct dati_privati *priv; /* lista dei campi ignota all'utente di priv */
  /* ... */
```

```
  struct dato *next; /* per l'inserimento in una lista */
};
```

7.7 ALLOCAZIONE DINAMICA DELLA MEMORIA

Quando si dichiara una variabile, il compilatore alloca automaticamente lo spazio in memoria necessario per memorizzare la variabile. La quantità di spazio allocato dipende dal tipo della variabile.

Quando si dichiara un puntatore ad un determinato tipo, viene allocato spazio in memoria per il puntatore soltanto, indipendentemente dalla dimensione del tipo puntato. Come si è visto, il puntatore potrà successivamente essere assegnato per contenere l'indirizzo di una variabile, e quindi si potrà utilizzare il puntatore per accedere al contenuto della variabile passando per il suo indirizzo.

Il linguaggio C permette di effettuare l'allocazione di memoria anche durante l'esecuzione del programma, sulla base della necessità e di opportune condizioni che possono verificarsi durante l'esecuzione. Questo tipo di allocazione di memoria è detta *dinamica*, proprio perché avviene dinamicamente durante l'esecuzione, mentre l'allocazione cosiddetta *statica* è quella che viene effettuata dal compilatore.

Allocare dinamicamente della memoria è una operazione molto comune nei programmi C. La possibilità di effettuare tale tipo di allocazione differenzia il C rispetto ad altri linguaggi, come Java, che non lo permettono. L'allocazione dinamica è una caratteristica molto potente a disposizione del programmatore, ma è anche spesso la sorgente di errori logici difficili da diagnosticare.

L'allocazione dinamica della memoria si effettua tramite la chiamata ad una funzione di libreria, che può essere utilizzata includendo il file di intestazione `<stdlib.h>`. La funzione `malloc`[18], la quale prende come parametro il numero di byte da allocare in memoria, e restituisce l'indirizzo al quale la memoria è stata allocata, oppure `NULL` se non è stato possibile allocare la memoria. Lo spazio allocato in memoria è contiguo. Un esempio dell'utilizzo della `malloc` è il seguente:

```
int *pt;
pt = malloc(10 * sizeof(*pt));
```

nel quale viene allocato lo spazio necessario per memorizzare 10 valori interi contigui, uno spazio di memoria che può quindi essere acceduto come fosse un vettore. Infatti è poi possibile utilizzare il puntatore indicizzandolo opportunamente per accedere alla memoria allocata, per esempio per azzerare tutti gli elementi interi memorizzati come segue:

```
for (i = 0; i < 10; i++)
  pt[i] = 0;
```

Tutta la memoria allocata dinamicamente deve essere rilasciata quando non più necessaria, per evitare di occupare inutilmente memoria che potrebbe essere necessario allocare in seguito. Per fare questo si chiama la funzione `free`, che accetta come parametro un puntatore contenente l'indirizzo della memoria da deallocare. Continuando l'esempio precedente, si scriverebbe

```
free(pt);
```

Chiamare la `free` su un puntatore non valido perché nullo o non precedentemente allocato con `malloc` può causare comportamenti imprevisti del programma.

Un esempio completo e più articolato di allocazione dinamica della memoria è dato dal seguente programma.

[18] `malloc` sta per *memory allocation*.

Il programma è contenuto nel file `alloc.c`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N (10)
5
6  struct t_strutt {
7    int num[N];
8    char *st;
9  };
10
11 int main() {
12   char s[80];
13   struct t_strutt *pt1, *pt2;
14   int i, j, n;
15   int min, max;
16
17   printf("numero di strutture da allocare: ");
18   fgets(s, sizeof(s), stdin);
19   sscanf(s, "%d", &n);
20
21   pt1 = malloc(n * sizeof(struct t_strutt));
22   if (pt1 == NULL) return -1;
23
24   for (i = 0; i < n; i++) {
25     min = RAND_MAX;
26     max = -RAND_MAX;
27     for (j = 0; j < N; j++) {
28       pt1[i].num[j] = rand() % 1000;
29       if (pt1[i].num[j] < min) min = pt1[i].num[j];
30       if (pt1[i].num[j] > max) max = pt1[i].num[j];
31     }
32     pt1[i].st = malloc(80);
33     sprintf(pt1[i].st, "memorizzati %d valori da %d a %d", N, min, max);
34   }
35
36   pt2 = pt1;
37
38   for (i = 0; i < n; i++) {
39     for (j = 0; j < N; j++) {
40       printf("%d ", pt2[i].num[j]);
41     }
42     printf("\n%s\n", pt2[i].st);
43   }
44
45   for (i = 0; i < n; i++) {
46     free(pt2[i].st);
47   }
```

```
48   free(pt2);
49
50   return 0;
51 }
```

Il programma gestisce delle variabili di tipo `struct t_strutt` la cui struttura è definita alle linee [6. . .9], contenente un vettore di `N` (pari a 10) interi ed un puntatore a carattere. Alle linee [17. . .19] viene richiesto all'utente il numero `n` di strutture da allocare. Un vettore di tali strutture viene allocato alla linea 21 con la chiamata alla `malloc`, e l'indirizzo è memorizzato nel puntatore `pt1`. Alla riga 22 si controlla che l'indirizzo ritornato sia non nullo. In caso di errore il programma termina. Il ciclo `for` che inizia a linea 24 inserisce dei dati all'interno delle `n` strutture, in particolare inserisce `N` valori interi casuali compresi tra 0 e 999 (linea 28). Questo è possibile grazie alla chiamata alla funzione di libreria `rand` per la generazione di numeri pseudo-casuali, e all'applicazione dell'operatore modulo (il simbolo di percentuale) che restituisce il resto della divisione intera. Le istruzioni 29 e 30 tengono traccia dei valori minimi e massimi generati; si noti a tal fine come sono stati opportunamente inizializzati i valori delle variabili `min` e `max`: `RAND_MAX` è definita nel file di intestazione `stdlib.h` con il valore costante che rappresenta il massimo numero intero che può essere generato dalla funzione `rand`. L'istruzione 32 alloca un vettore di 80 caratteri e ne assegna l'indirizzo al puntatore a carattere contenuto nella struttura. Questo vettore viene utilizzato dall'istruzione 33 per memorizzarvi una stringa. La funzione `sprintf`, infatti, opera come la nota `printf`, ma invece di scrivere a video scrive la stringa di output all'indirizzo specificato dal primo parametro, che non è altro che il vettore appena allocato.

L'istruzione 36 assegna a `pt2` l'indirizzo contenuto in `pt1`. Questo assegnamento ha scopo puramente didattico, e serve per mostrare come d'ora in poi si possa utilizzare `pt2` per accedere a tutti i dati finora gestiti tramite il puntatore `pt1`.

Il ciclo annidato alla linea 38 visualizza a video il contenuto di tutte le strutture stampando, per ciascuna struttura, prima il contenuto del vettore di interi e poi quello della stringa.

Al termine del programma, dalla linea 45, tutte le variabili sono deallocate mediante la chiamata a `free`. In questa fase, è importante notare che prima viene rilasciata la memoria allocata per le stringhe, e poi si rilasca la memoria allocata al vettore di strutture. È importante osservare questa sequenza nel rilascio della memoria. Se infatti si rilasciasse prima il puntatore associato al vettore di strutture, ci si troverebbe ad utilizzare un'area di memoria che nel frattempo potrebbe essere stata utilizzata dal sistema operativo per altri scopi, e quindi non conterrebbe più dati validi. Un puntatore in questo stato si dice `puntatore dangling`.

Un semplice modo per evitare di utilizzare erroneamente puntatori dangling è quello di assegnare esplicitamente al puntatore il valore `NULL` dopo la chiamata alla funzione `free`, come evidenziato nell'esempio seguente:

```
1  #include <stdlib.h>
2  {
3    char *cp = malloc ( A_CONST );
4
5    /* ... */
6    free ( cp );
7
8    /* cp diviene un puntatore dangling */
9    cp = NULL;
10
11   /* ora cp non e' piu' dangling */
12   /* ... */
13 }
```

Dopo l’assegnamento, un tentativo di utilizzare il puntatore `cp` genera un errore di accesso alla memoria (segmentation fault), e diviene così più semplice scoprire eventuali accessi errati.

Capitolo 8

FUNZIONI

Una *funzione* in C è una porzione di codice, detto anche *sottoprogramma* o, in inglese, *subroutine*, che può essere *richiamata più volte* in un programma.

Padroneggiare l'uso delle funzioni è di fondamentale importanza nella scrittura di programmi, in particolare quando la complessità dei programmi aumenta. Quando una serie di operazioni devono essere eseguite varie volte all'interno del programma, magari al variare di certi parametri è molto consigliabile usare una funzione che implementa le operazioni una volta per tutte, e chiamare la funzione quando è necessario eseguire le relative istruzioni.

L'uso delle funzioni aumenta notevolmente la chiarezza e la modularità di un programma, nonché la riutilizzabilità del codice prodotto. Le funzioni possono essere scritte dal programmatore ed utilizzate in un programma, oppure possono essere reperite in *librerie di codice* di utilità generale, ovvero collezioni di funzioni già implementate a disposizione per la soluzione di problemi comuni.

Ogni funzione ritorna un solo valore, di un tipo semplice o una struttura dati, oppure `void`, cioè nulla, e riceve uno o più argomenti.

In un programma ci possono essere *dichiarazioni di funzione* e *definizioni di funzione*. Almeno una funzione è sempre presente: `main`.

Ogni nome di funzione può essere presente una volta sola in un programma e ogni chiamata deve passare sempre lo stesso numero e tipo di argomenti[19]. A questa regola fanno eccezione le cosiddette *funzioni variadiche* (Sezione 8.8), alle quali possono essere passati un numero arbitrario argomenti ulteriori a quelli predefiniti, eventualmente anche 0.

[19]Non esiste il polimorfismo delle funzioni in C, concetto fondamentale nei linguaggi di programmazione ad oggetti come C++ o Java.

NOTA — Le funzioni disponibili nelle librerie sono solitamente ottimizzate, cioè realizzate nel miglior modo possibile, ed è quindi sempre buona prassi utilizzare delle funzioni di libreria quando disponibili invece che duplicarne l'implementazione.

8.1 DICHIARAZIONE DI FUNZIONI

In una dichiarazione di funzione (*prototyping*) si riconoscono:

- il tipo di dato restituito dalla funzione, oppure `void` se la funzione non restituisce alcun valore; se non viene specificato il valore di ritorno, si assume che la funzione ritorni `int`,
- il nome della funzione
- i tipi degli eventuali argomenti, `void` o niente se la funzione non usa parametri
- la dichiarazione termina con ';'

Nella dichiarazione non si specificano le istruzioni che compongono la funzione, ma soltanto il suo *prototipo* o *interfaccia* o , cioè il nome, l'elenco degli argomenti e il tipo restituito. Una dichiarazione serve per informare in compilatore che la funzione, comprensiva delle relative istruzioni, è definita da qualche altra parte del codice. In questo modo, quando il compilatore incontra una eventuale chiamata a funzione in un punto del codice nella quale tale funzione non è ancora stata definita, il compilatore può effettuare tutti i controlli necessari sulla correttezza della chiamata, grazie al fatto che gli è stata resa nota l'interfaccia della funzione.

Per un corretto funzionamento del programma la dichiarazione di una funzione deve sempre precedere l'invocazione della stessa, mentre la definizione può essere presente in un qualunque punto del sorgente.

Il seguente codice:

```
void funzione1(int arg1, double arg2);
```

dichiara la funzione `funzione1` che non restituisce alcun valore, ovvero restituisce `void`, e richiede due argomenti, il primo dei quali è intero ed il secondo è un reale in doppia precisione.

Nell'esempio seguente:

```
char funzione2(void);
```

viene dichiarata la funzione `funzione2` che restituisce un carattere e non richiede alcun parametro.

8.2 DEFINIZIONE DI FUNZIONI

Una definizione di funzione è costituita da due parti:

1. una dichiarazione, in cui sono elencati:
   - il tipo di dato restituito dalla funzione
   - il nome della funzione
   - gli eventuali argomenti
2. il *corpo della funzione*, racchiuso tra parentesi graffe e comprendente queste parti opzionali:
   - dichiarazioni e definizioni di variabili
   - istruzioni

- una o più istruzioni `return`

Si è affermato che tutte le parti che compongono il corpo di una funzione sono opzionali. Infatti, una funzione come la seguente

```
do_nothing(){ }
```

ovvero il cui corpo non comprende alcuna istruzione, è perfettamente lecita.

8.3 STRUTTURAZIONE DEL PROGRAMMA IN FUNZIONI

Un programma, per quanto semplice esso sia ed escludendo i casi non interessanti di programmi privi di istruzioni, è praticamente sempre suddivisibile in funzioni.

Nei capitoli precedenti sono stati fatti vari esempi di programmi, nei quali tutte le istruzioni venivano inserite nella funzione `main`. Questo è stato fatto non perché tali programmi non potessero essere scritti facendo uso di funzioni, ma semplicemente perché il concetto e l'utilizzo di funzioni é stato introdotto soltanto ora.

Per mostrare come molti degli esempi effettuati in precedenza possano essere riscritti in forma modulare utilizzando le funzioni, riprendiamo l'esempio riportato alla Sezione 5.4. Il programma effettua il calcolo di una serie. Qui di seguito è riportata la versione dello stesso programma realizzata mediante l'uso di funzioni. In effetti il programma che segue è stato leggermente esteso per mostrare come, utilizzando le funzioni, è possibile eliminare la duplicazione non necessaria di codice.

Il programma è contenuto nel file `somma_func.c`.

```
1  #include <stdio.h>
2
3  long ottieni_limite();
4  double serie(long lim);
5
6  int main()
7  {
8    long max;
9    double val;
10
11   max = ottieni_limite();
12   val = serie(max);
13   printf("la somma e' %f\n", val);
14
15   max = ottieni_limite();
16   val = serie(max);
17   printf("la somma e' %f\n", val);
18
19   return 0;
20 }
21
22 long ottieni_limite()
23 {
24   long max;
25
```

```
26    printf("valore massimo di i: ");
27    scanf("%ld", &max);
28
29    return max;
30  }
31
32  double serie(long lim)
33  {
34    double somma;
35    int i;
36
37    for(i = 1, somma = 0.0; i <= lim; i++)
38      somma += 1.0 / (i * i * i);
39
40    return somma;
41  }
```

Sommariamente, sono da notare le seguenti caratteristiche del programma in questa versione "modularizzata":

- prima della funzione `main`, alle linee 3 e 4 è stata effettuata la dichiarazione delle funzioni;
- le funzioni vengono richiamate nel `main`;
- le funzioni vengono poi definite dopo il `main` stesso, iniziando rispettivamente alle linee 22 e 32; questa prassi è molto comune;
- nel `main` le due funzioni vengono richiamate due volte, e questo mostra, anche in questo semplice esempio, come l'aver posto il codice all'interno di funzioni permetta di evitare di duplicare codice;
- alla funzione `serie` viene passato come parametro il numero di iterazioni da effettuare;
- l'aver spostato il codice dal `main` alle funzioni ha *ridotto* il numero di variabili dichiarate nel `main` stesso; le variabili che servono solo all'interno delle funzioni sono dichiarate localmente alle funzioni stesse, e questo evita di avere troppe variabili leggibili/scrivibili in punti del programma che non ne necessitano (si veda il Capitolo 10 per le regole che determinano la cosiddetta *visibilità* delle variabili).

8.4 PASSAGGIO DEI PARAMETRI

Una funzione può utilizzare parametri per svolgere il proprio compito, cioè delle variabili i cui valori iniziali vengono passati alla funzione al momento della sua invocazione e che contengono valori necessari a controllare le operazioni svolte dalla funzione, oppure sono valori utilizzati nei calcoli svolti dalla funzione.

La comunicazione fra il codice chiamante ed il sottoprogramma avviene sempre secondo la modalità del *passaggio per valore*. Questo significa che la funzione utilizza una copia locale della variabile passata come parametro all'atto della chiamata, e tale variabile locale è inizializzata con il valore dell'espressione o della variabile impostata nel punto di chiamata alla funzione. In particolare, la funzione potrà modificare il valore della variabile locale, ma non quello dell'eventuale variabile passata come parametro all'invocazione della funzione stessa, in quanto le due variabili *sono allocate in aree di memoria distinte*.

Un esempio di passaggio dei parametri ad una funzione creata dall'utente è riportato nell'esempio seguente:

Il programma è contenuto nel file `func_decl.c`.

```
1  #include <stdio.h>
2
3  float massimo(float, float);
4
5  int main()
6  {
7    int a, b;
8
9    scanf("%d %d", &a, &b);
10   printf("il massimo e': %f", massimo(a, b));
11   return 0;
12 }
13
14 /* definizione di funzione */
15 float massimo(float a, float b)
16 {
17   return b > a ? b : a;
18 }
```

Il programma dichiara (linea 3) e definisce (da linea 15) la funzione `massimo` per il calcolo del massimo tra due numeri in virgola mobile. Un esempio di utilizzo della funzione viene fatto alla linea 10, all'interno della `printf`, per calcolare il massimo tra le variabili `a` e `b` precedentemente inseriti dall'utente. Le variabili `a`, `b` della funzione `massimo` non hanno alcun legame con le variabili `a` e `b` dichiarate all'interno della funzione `main`. Infatti, l'invocazione

```
massimo(b, a);
```

era altrettanto lecita. Si noti che nel `main` sono definite come `int` mentre nella funzione come `float`: il compilatore effettua automaticamente le conversioni di tipo necessarie.

Il calcolo del massimo fa uso dell'operatore ternario "? :" (si veda la Sezione 9.29 per maggiori dettagli), alla linea 17 il quale valuta la condizione che precede il punto di domanda, ovvero `b > a`, e nel caso essa sia vera ritorna il valore di `b`, altrimenti quello di `a`.

8.5 PASSAGGIO PER RIFERIMENTO

In molte situazioni è necessario permettere alla funzione chiamata di poter modificare il valore della variabile passata dal chiamante all'atto dell'invocazione della funzione. La tecnica per il passaggio dei parametri che permette questa operazione è detta *passaggio per riferimento*. Il passaggio per riferimento implica il passaggio del *puntatore alla variabile*. In questo modo alla funzione chiamata è nota la locazione di memoria alla quale la variabile di interesse è allocata, ed è quindi possibile modificare il valore della variabile originale scrivendo direttamente in tale locazione di memoria.

NOTA Tecnicamente il linguaggio C permette soltanto il passaggio per valore. Il passaggio per riferimento viene realizzato quindi passando per valore l'indirizzo di una variabile.

Per esempio, una funzionalità spesso utile è quella dello scambio del valore di due variabili. Questo può essere fatto implementando una funzione apposita che, come nell'esempio seguente, scambia il valore di due interi passati per riferimento:

```
void swap(int *a, int *b)
{
  int tmp;

  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Gli argomenti delle funzioni possono essere tipi semplici o strutture dati e, come detto, sono sempre passati per valore; anche se è consentito, normalmente non vengono passate strutture dati né come argomenti né come valori di ritorno. Si preferisce invece, per ragioni di efficienza, allocare le strutture dati separatamente e passare solo i puntatori ad esse, effettuando così un passaggio per riferimento.

Perché si parla di questioni di efficienza? Come detto, il passaggio dei parametri che avviene per valore comporta l'allocazione di una copia locale delle variabili dichiarate nella lista dei parametri. Oltre all'allocazione, tali variabili devono anche essere inizializzate per riflettere il valore delle variabili o espressioni del chiamante. Questo comporta, nel caso in cui il parametro passato sia una variabile, la copia esplicita di una porzione di memoria dalla variabile utilizzata per la chiamata alla variabile locale. Nel caso le variabili siano strutture dati c'è quindi una perdita di efficienza nel passaggio dei parametri che è proporzionale alla dimensione della variabile, in quanto il tempo necessario alla copia del valore aumenta all'aumentare della dimensione della struttura.

Un altro caso in cui è utilizzato il passaggio per riferimento è nel caso in cui una funzione debba ritornare più di un valore, per esempio un numero intero e un codice di errore. Anche in questo caso è possibile utilizzare il passaggio per riferimento e passare quindi dei puntatori come argomenti ulteriori, in modo che la funzione possa scrivere i valori di ritorno in una o più variabili del chiamante.

8.6 LA FUNZIONE `MAIN`

Un programma C deve sempre includere una funzione chiamata `main`. La funzione `main`, o "il `main`", è il punto dal quale il programma inizia la sua esecuzione. La funzione `main` può essere definita in due modi. Il primo prevede di specificare il tipo restituito, che deve essere sempre intero, e nessun parametro. L'esempio è il seguente:

```
int main(void)
```

Si noti che in taluni contesti è possibile trovare definizioni del tipo

```
void main(void)
```

Tale definizione viene accettata dal compilatore, con la generazione di un warning, ma dovrebbe essere evitata e in queste dispense sarà considerata una definizione errata.

L'altra possibilità è quella di dotare il `main` di due parametri, come nell'esempio seguente

```
int main(int argc, char **argv)
```

I parametri verranno inizializzati all'atto della chiamata della funzione `main`, ovvero dell'avvio del programma, con i valori recuperati dalla linea di comando utilizzata per invocare il programma stesso.

Il semplice programma seguente stampa a video alcune informazioni relative alla linea di comando.

Il programma è contenuto nel file `cmd_line.c`.

```
1  #include <stdio.h>
2
3  int main(int argc, char ** argv)
4  {
5    int i;
6
7    for (i = 0; i < argc; i++) {
8      printf("Argomento %d (%d): %s\n", i, argc, argv[i]);
9    }
10
11   return 0;
12 }
```

Dopo aver compilato il programma con il comando

```
$ cc -Wall -o cmd_line cmd_line.c
```

si provi a verificare l'output ottenuto invocando il programma appena generato. Per esempio, si otterrà:

```
$ ./cmd_line
Argomento 0 (1): ./cmd_line

$ ./cmd_line arg1 arg2 arg3
Argomento 0 (4): ./cmd_line
Argomento 1 (4): arg1
Argomento 2 (4): arg2
Argomento 3 (4): arg3

$ ./cmd_line arg1    arg2      arg3
Argomento 0 (4): ./cmd_line
Argomento 1 (4): arg1
Argomento 2 (4): arg2
Argomento 3 (4): arg3

$ ./cmd_line "arg1    arg2      arg3"
Argomento 0 (2): ./cmd_line
Argomento 1 (2): arg1    arg2      arg3
```

In particolare, si noti l'effetto (nullo) di spazi extra inseriti tra gli argomenti nel terzo esempio. Mentre l'effetto degli apici nel quarto esempio fa si che il programma ottenga un unico parametro inclusivo degli spazi.

8.7 PUNTATORI A FUNZIONE

L'argomento dei puntatori a funzione viene trattato in quanto i puntatori a funzione sono molto usati in programmi complessi scritti da bravi programmatori. L'argomento viene però trattato in modo non molto approfondito poiché, se da un lato l'uso dei puntatori rende spesso molto chiaro, compatto ed elegante il codice che ne fa uso, dall'altro rappresenta un argomento piuttosto avanzato rispetto a quelli che vengono trattati in questo documento.

Come nel caso dei vettori, una funzione viene rappresentata dall'indirizzo del codice associato. Tutte le volte che si usa un nome di funzione in un programma si sta in pratica usando il puntatore

a tale funzione. La chiamata a funzione, semplificando un po', consiste in pratica nel recuperare l'indirizzo della funzione da eseguire (quindi un puntatore) e nell'impostare il Program Counter per iniziare ad eseguire le istruzioni che sono presenti all'indirizzo specificato dal puntatore.

Un puntatore a funzione viene dichiarato come segue:

```
nome_tipo ( * nome_variabile ) ( <parametri> ) ;
```

La sintassi precedente dichiara una variabile chiamata `nome_variabile`, la quale è un puntatore a funzione alla quale possono essere assegnati indirizzi di funzioni che restituiscono un tipo specificato da `nome_tipo` e che richiede un opportuno numero e tipo di parametri. Si noti che la dichiarazione di un puntatore a funzione richiede esplicitamente che il simbolo di asterisco e il nome della variabile siano racchiusi tra parentesi tonde. Infatti, la scrittura

```
nome_tipo * nome_variabile ( <parametri> ) ;
```

non dichiara un puntatore a funzione bensì una funzione che ritorna un puntatore a `nome_tipo`, che è cosa ben diversa.

Per esempio, le due dichiarazioni seguenti:

```
double sommatoria(int n);
double (*operazione)(int n);
```

dichiarano rispettivamente una funzione ed un puntatore a funzione. La dichiarazione di un puntatore a funzione non necessita di nient'altro per permettere al programma che lo dichiara di compilare correttamente: si tratta semplicemente di una variabile che può essere utilizzata o meno. Per contro, la dichiarazione di una funzione, quindi non di una variabile puntatore a funzione, richiede anche la definizione del suo corpo, cioè delle istruzioni che la compongono.

Il seguente programma presenta un semplice esempio di uso di puntatori a funzione.

Il programma è contenuto nel file `ptr_funzione.c`.

```
1  #include<stdio.h>
2
3  int somma(int, int);
4  int sottrazione(int, int);
5  int prodotto(int, int);
6  int divisione(int, int);
7
8  int main()
9  {
10   int a = 12, b = 4;
11   int c, scelta;
12   int (*operazione)();
13
14   while (1) {
15     printf("1) somma\n");
16     printf("2) sottrazione\n");
17     printf("3) prodotto\n");
18     printf("4) divisione\n");
19     printf("0) uscire\n");
20
21     scanf("%d", &scelta);
```

```
22
23      switch(scelta) {
24        case 1:
25          operazione = somma;
26          break;
27        case 2:
28          operazione = sottrazione;
29          break;
30        case 3:
31          operazione = prodotto;
32          break;
33        case 4:
34          operazione = divisione;
35          break;
36        case 0:
37          return 0;
38      }
39
40      c = operazione(a, b);
41      printf("--- Il risultato vale %d\n", c);
42    }
43  }
44
45  int somma(int a, int b)
46  {
47    return a + b;
48  }
49
50  int sottrazione(int a, int b)
51  {
52    return a - b;
53  }
54
55  int prodotto(int a, int b)
56  {
57    return a * b;
58  }
59
60  int divisione(int a, int b)
61  {
62    return (a / b);
63  }
```

Nell'esempio ci sono diverse funzioni, aventi la stessa interfaccia (tipo restituito, numero e tipo degli argomenti) che effettuano diverse operazioni aritmetiche sugli argomenti. Una variabile puntatore a funzione viene assegnata a seconda della scelta dell'utente, e la funzione scelta viene chiamata mediante il puntatore. In questo modo risulta chiaro come viene dichiarato un puntatore a funzione, come viene assegnato ed utilizzato per richiamare una funzione.

L'uso più consueto di un puntatore a funzione è l'applicazione dell'operatore parentesi-tonde, situazione che normalmente non viene pensata in termini di puntatori ed operatori da parte del programmatore, ma semplicemente in termini di chiamata a funzione. Le parentesi tonde che racchiu-

dono l'elenco degli argomenti della funzione, non sono altro che un operatore applicato all'operando costituito dal nome della funzione, che è un puntatore.

Un puntatore a funzione può anche essere assegnato ad altri puntatori, per esempio all'interno di strutture dati che definiscono i metodi con cui operare sugli oggetti, oppure passato come argomento ad altre funzioni, per esempio la funzione di libreria `qsort`, funzione che implementa l'algoritmo "quick sort" per l'ordinamento di un vettore generico di elementi. Il compilatore verifica in compilazione che i tipi dei puntatori a funzione siano compatibili, cioè le funzioni ricevano gli argomenti corretti, compatibilmente a come sono state dichiarate.

8.8 NUMERO VARIABILE DI PARAMETRI

Si possono definire funzioni con numero variabile di argomenti (*variadiche*). L'esempio più comune di funzione variadica è `printf` con tutte le sue varianti. Definire la propria funzione variadica richiede una certa attenzione e conoscenza del meccanismo di passaggio dei parametri, e non verrà quindi trattato in questa sede.

Chiamare una funzione variadica è invece molto frequente e basta specificare correttamente tutti gli argomenti. Nel caso dei derivati di `printf`, uno dei primi argomenti è una stringa che specifica il numero e il tipo degli argomenti successivi. La funzione variadica usa la stringa per sapere cosa sono gli argomenti ulteriori; data la standardizzazione del formato della stringa, il compilatore può controllare tutti gli argomenti passati e avvertire del possibile errore in caso di incongruenze. Per funzioni variadiche non assimilabili a `printf` il controllo del compilatore non è previsto. Una funzione si intende assimilabile a `printf` quando utilizza lo stesso formato per specificare il tipo dei parametri.

Capitolo 9

GLI OPERATORI

Gli operatori nel linguaggio C si dividono principalmente in operatori aritmetici, logici e operatori che agiscono sui bit. In questo capitolo verranno descritti nel dettagli tutti gli operatori a disposizione, mentre una tabella riassuntiva degli operatori è riportata in Appendice 9.

Una caratteristica importante degli operatori riguarda il loro *ordine di precedenza*, ovvero con quale ordine vengono applicati quando ve ne sia più d'uno all'interno della stessa espressione. Per esempio nell'espressione aritmetica

```
a = 10 * 2 + 7 * 3
```

la variabile `a` assume il valore 41, in quanto l'operatore "*" ha precedenza maggiore rispetto all'operatore "+".

Un'altra caratteristica importante è costituita dal numero di operandi richiesti da ciascun operatore. A seconda dell'operatore, potranno essere necessari 1, 2 o 3 operandi.

È da tenere comunque presente che l'ordine di precedenza può sempre essere modificato con l'uso opportuno delle parentesi tonde. Nell'esempio precedente, è possibile scrivere:

```
a = 10 * (2 + 7) * 3
```

il cui risultato è 270.

In questo capitolo verranno illustrati tutti gli operatori disponibili nel linguaggio C, nello stesso ordine di precedenza indicato in Appendice 9.

9.1 PRECEDENZA DEGLI OPERATORI

Per precedenza si intende l'ordine con il quale vengono considerate le espressioni collegate dagli operatori.

Nell'esempio

```
espr1 op1 espr2 op2 espr3
```

L'espressione è composta da più operatori che nel caso in cui non siano usate le parentesi verranno applicati a seconda del loro ordine di precedenza (l'ordine di precedenza di tutti gli operatori è desubimibile dalla Tabella in Appendice A). Se `op1` ha precedenza maggiore di `op2`, viene prima valutata l'espressione `espr1 op1 espr2`, ed il risultato viene composto a destra con `espr3`; il risultato è quindi equivalente a

```
(espr1 op1 espr2) op2 espr3
```

se invece `op1` ha precedenza minore di `op2`, viene prima valutata l'espressione `espr2 op2 espr3`, ed il risultato viene composto a sinistra con `espr1`, ed il risultato è invece equivalente a

```
espr1 op1 (espr2 op2 espr3)
```

9.2 ORDINE DI VALUTAZIONE

L'ordine di valutazione delle espressioni sulle quali un operatore viene applicato non è specificato dal linguaggio C, ovvero tale ordine è *indefinito*.

Se `espr1` e `espr2` sono sotto-espressioni composte con l'operatore `op`, come nell'esempio

```
  espr1 op espr2
```

allora si ha che:

- l'ordine di valutazione di `espr1` e `espr2` è indefinito
- se viene valutata prima `espr1` o prima `espr2` dipende dal compilatore

Un esempio è riportato nella Sezione 9.4.

9.3 IL CONCETTO DI SIDE EFFECT

Si definisce "side effect" (o effetto collaterale) il risultato di un operatore, espressione, o funzione che persiste dopo la sua valutazione.

Nell'esempio:

```
x = 10;
```

il side effect consiste nel fatto che dopo la valutazione dell'espressione il valore di `x` cambia permanentemente in 10.

9.4 SIDE EFFECT E ORDINE DI VALUTAZIONE

L'ordine indefinito di valutazione delle espressioni può causare problemi quando più di un operatore che causa un *side effect* è utilizzato nella stessa espressione:

Nell'esempio seguente:

```
i = 0;
c = a[i] + b[++i];
```

il risultato della seconda espressione può essere

```
c = a[0] + b[1]
```

oppure

```
c = a[1] + b[1]
```

a seconda del compilatore che viene utilizzato.

Per questo motivo è bene evitare di usare espressioni che producano effetti collaterali su variabili utilizzate anche in altre parti della stessa espressione.

9.5 ASSOCIATIVITÀ DEGLI OPERATORI

L'associatività indica in quale ordine sono considerati gli operatori aventi la stessa priorità.

Sia l'operatore di dereferimento `*` che l'operatore di incremento `++` (sia che venga impiegato in forma postfissa, che in forma prefissa), hanno la stessa priorità nelle precedenze degli operatori, ma la loro associatività va da destra a sinistra

```
int vet[5];
int *p;

p = vet;
*p++ = 10;

p = vet;
*++p = 10;

p = vet;
*p = 10;
vet[1] = ++*p;
```

L'espressione `*p++ = 10` equivale a `vet[0] = 10`, mentre `*p` equivale a `vet[1]`. L'espressione viene interpretata come `*(p++)`: l'operatore incremento (postfisso) è applicato alla variabile `p` (puntatore): l'effetto è quello dell'utilizzo dell'area di memoria puntata dal puntatore `p` (`*p`) ed in seguito viene effettuato l'incremento del puntatore (`p++`).

L'espressione `*++p = 10` (`vet[1] = 10`; `*p` equivale a `vet[1]`) è interpretata come `*(++p)`: l'operatore di incremento è applicato a `p` in forma prefissa, quindi prima viene incrementato il puntatore `p (++p)`, e poi (con il valore aggiornato del puntatore) viene effettuato l'accesso all'area di memoria puntata da `p` incrementato.

Nell'ultima espressione `vet[1] = ++*p` (`vet[1] = 11`; `*vet[0] = 11`; `*p` equivale a `vet[0]`), `++*p` è equivalente a `++(*p)`, cioè l'area di memoria puntata da `p` viene acceduta subito con l'operatore di indirizzamento `*`; in seguito, l'area di memoria viene incrementata dall'operatore di incremento `++` in forma prefissa, e il valore risultante viene poi usato nell'assegnamento.

Con riferimento al primo esempio, è da notare che l'espressione `(*p)++`, la quale incrementa il contenuto `(*p)` a cui punta `p`, non può essere scritta se non con l'uso delle parentesi. Inoltre, l'istruzione

```
(*p)++ = 10;
```

genera un errore di compilazione.

Per una maggiore chiarezza di lettura del codice, soprattutto in fase di manutenzione di un programma, è consigliabile utilizzare le parentesi per evidenziare l'associatività degli operatori, per evitare di dover ricorrere alle regole sintattiche del linguaggio.

9.6 CHIAMATA A FUNZIONE

operatore	( ) parentesi tonde
sintassi	`nome_funzione` (`parametri`)
n. operandi	1
utilizzo	chiamata della funzione `nome_funzione`
associatività	$\Longrightarrow$
commutatività	NO

L'operatore di chiamata a funzione si applica ad un solo operando: un nome di funzione o un puntatore, specificando gli argomenti da passare dentro le parentesi, ognuno dei quali è una espressione.

Nell'istruzione

```
double num = sqrt(16);
```

le parentesi tonde indicano la chiamata della funzione di libreria `sqrt`, che delimitano l'elenco dei parametri che in questo caso è uno solo, il valore costante 16.

L'operatore parentesi-tonde si applica anche ai puntatori a funzione:

```
int (*operazione)(int a, int b);
int num = operazione(12, 4);
```

9.7 ELEMENTO DI VETTORE

operatore	[ ] parentesi quadre
sintassi	`vett` [`id`]
n. operandi	2
utilizzo	accesso all'elemento di indice `id` del vettore `vett`
associatività	$\Longrightarrow$
commutatività	SI'

Serve per indirizzare l'elemento di un vettore. L'operatore accetta due operandi, uno prima delle parentesi e l'altro tra parentesi; normalmente il primo è un puntatore e il secondo un numero intero, anche se in realtà l'operazione è commutativa. Dato il puntatore ad intero (o vettore) `v`, le seguenti istruzioni sono tutte equivalenti:

```
v[3] = 0;
*(v+3) = 0;
3[v] = 0;
```

9.8 ELEMENTO DI STRUTTURA

operatore	. punto
sintassi	`strutt . camp`
n. operandi	2
utilizzo	accesso al campo `camp` della struttura `strutt`
associatività	$\Longrightarrow$
commutatività	NO

Viene utilizzato per indicare l'elemento di una struttura. I due operandi sono una struttura, ovvero un'espressione il cui valore è una struttura, e il nome di campo di tale struttura.

Nell'esempio seguente, `x` è un campo di tipo intero della struttura `struct t_punto`:

```
struct t_punto {
  int x;
  int y;
};

struct t_punto st, *stptr = &st, stvec[10];
int a, b, c;

a = st.x;
b = (*stptr).x;
c = stvec[5].x;
```

Si noti che il puntatore `stptr` deve essere allocato o assegnato prima di utilizzarlo. Inoltre, l'utilizzo dell'operatore "." che indica il campo specifico della struttura `struct t_punto`.

9.9 ELEMENTO DI STRUTTURA DA PUNTATORE

operatore	$->$ freccia (trattino alto + maggiore)
sintassi	`strutt_ptr -> camp`
n. operandi	2
utilizzo	accesso al campo `camp` della struttura puntata da `strutt_ptr`
associatività	$\Longrightarrow$
commutatività	NO

È usato quando si deve indicare un elemento di una struttura puntata da un puntatore. I due operandi sono un puntatore a struttura e il nome di un campo di tale struttura. È il modo più comune per utilizzare le strutture dati.

```
struct t_punto {
  int x;
  int y;
};

struct t_punto st, *stptr, stvec[10];
int a;

/* funzione che ritorna un puntatore a struct t_punto */
struct t_punto *get_punto(int i);

a = stptr->x;
a = (stvec + 5)->x;
a = (&st)->x;   /* & tra parentesi, -> ha priorita' maggiore di & */
get_punto(5)->x;
```

9.10 NEGAZIONE LOGICA

operatore	! punto esclamativo
sintassi	! `espr`
n. operandi	1
utilizzo	negazione logica dell'espressione `espr`
associatività	$\Longleftarrow$
commutatività	NO

L'operatore di negazione logica nega l'operando alla sua destra: se l'operando è 0 il risultato è 1, se l'operando è non-zero il risultato è zero.

Nel seguente esempio, la funzione `malloc` ritorna `NULL` se non riesce ad allocare il puntatore. In tal caso il valore di `p` è nullo, quindi la sua negazione corrisponde al valore logico vero, che è quanto serve per gestire la condizione di errore.

```
p = malloc(128);
if (!p) {
  /* gestione errore */
}
```

Nell'esempio seguente, invece, la doppia negazione fa si che il valore di ritorno valga 0 se `val` vale 0, mentre vale 1 se `val` è diverso da zero.

```
return !!val;
```

Per esempio, si ha:

```
!!3 = 1;
!!0 = 0;
```

Questo espediente è talvolta utilizzato in quanto, si ricordi, che il valore zero è associato al valore logico "falso", mentre ogni valore diverso da zero è associato al valore logico "vero". In questo modo si restringe l'insieme dei valori logici ai due vallori uno e zero.

9.11 COMPLEMENTO A 1

operatore	$\sim$ tilde
sintassi	$\sim$ `val`
n. operandi	1
utilizzo	complemento a 1, corrispondente alla negazione dei singoli bit di `val`
associatività	$\Longleftarrow$
commutatività	NO

Si tratta di un operatore logico che nega i bit dell'operando alla sua destra, ovvero pone a 1 tutti i bit che sono a 0 e viceversa.

```
/* valore di i: 0xffffffff se 32 bit, 0xff se 8 bit, eccetera */
i = ~0;
```

9.12 NEGAZIONE UNARIA

operatore	$-$ trattino alto – segno meno
sintassi	$-$ `espr`
n. operandi	1
utilizzo	cambia segno all'espressione `espr`
associatività	$\Longleftarrow$
commutatività	NO

La negazione unaria nega l'espressione aritmetica alla sua destra, cioè cambia di segno al risultato dell'espressione.

```
i = -1;

/* EINVAL e' un codice di errore intero positivo */
return -EINVAL;
```

9.13 INCREMENTO E DECREMENTO

operatore	++
	più più
sintassi	`val++` oppure `++val`
n. operandi	1
utilizzo	incrementa di una unità il valore di `val`
associatività	⟸
commutatività	NO

operatore	−−
	meno meno
sintassi	`val−−` oppure `−−val`
n. operandi	1
utilizzo	decrementa di una unità il valore di `val`
associatività	⟸
commutatività	NO

Si tratta di operatori con un solo operando; se l'operatore sta dopo l'operando (esempio: `i++`) l'incremento o decremento viene fatto dopo aver usato il valore dell'operando, se l'operatore sta prima dell'operando (esempio: `++i`), l'incremento o decremento viene fatto prima di usare il valore. L'operando deve essere assegnabile (si veda la discussione su "lvalue" nella Sezione 9.30 sull'operatore di assegnamento).

Nell'esempio seguente gli operatori di incremento e decremento sono utilizzati per la semplice gestione di uno stack di numeri interi:

```
int stack[10]; /* uno stack di 10 elementi interi */
int sp = 0;    /* indice del primo elemento vuoto */

/* inserimento ("push") di un valore nello stack */
stack[sp++] = dato;

/* estrazione ("pop") di un valore dallo stack */
dato = stack[--sp];
```

Nell'esempio seguente, invece, a seconda che venga usato l'operatore di decremento nella sua forma prefissa piuttosto che postfissa, cambia il range nel quale varia il valore della variabile `i` che controlla il ciclo.

```
i = 10; while (--i) {
  /* ciclo per i che varia da 9 a 1 */
}
i = 10; while (i--) {
```

```
  /* ciclo per i che varia da 9 a 0 */
}
```

9.14 ESTRAZIONE DI UN INDIRIZZO

operatore	& “e” commerciale
sintassi	`& espr`
n. operandi	1
utilizzo	ritorna l’indirizzo di `espr`
associatività	$\Longleftarrow$
commutatività	NO

Esempio:

```
#include <sys/stat.h>
struct stat stbuf;

/* la funzione chiamata scrive in stbuf */
stat("/bin/sh", &stbuf);

char s[32];

/* assegna a i il valore intero contenuto nella stringa s */
sscanf(s, "%i", &i);
```

9.15 USO DI PUNTATORE

operatore	* asterisco
sintassi	`* ptr`
n. operandi	1
utilizzo	dereferenzia il puntatore `ptr`
associatività	$\Longleftarrow$
commutatività	NO

L’operatore permette l’uso del valore contenuto all’indirizzo memorizzato in un puntatore. L’esempio seguente effettua un ciclo che somma tutti gli elementi del vettore `v`.

```
int v[32], *p, somma = 0;
for (p = v + 31; p >= v; p--)
  somma = somma + *p;
```

Il puntatore `p` viene inizialmente fatto puntare all'ultimo elemento del vettore, e ad ogni iterazione il puntatore viene fatto puntare all'elemento precedente, finché l'indirizzo in esso memorizzato è maggiore o uguale all'indirizzo del primo elemento del vettore (`p >= v`). Si ricordi infatti che il nome di un vettore senza le parentesi quadre indica l'indirizzo del primo elemento del vettore. Ad ogni iterazione viene sommato al valore di `somma` il valore puntato da `p`, cioè `*p`.

9.16 OPERATORE DI CASTING

Il *casting* di una variabile significa cambiare il tipo di dato associato alla variabile stessa.

operatore	( ) parentesi tonde
sintassi	`( type ) espr`
n. operandi	2
utilizzo	cambia il tipo di `espr` in `type`
associatività	⟸
commutatività	NO

Spesso il casting non comporta generazione di codice, per esempio per convertire tra `unsigned` e `signed`, o tra puntatori di tipo diverso.

```
/* mmap ritorna un indirizzo e
 * l'indirizzo -1 indica errore
 */
addr = mmap( /* argomenti */ );
if (addr == (void *)-1) { /* gestione errore */ }
```

9.17 DIMENSIONE DI UNA VARIABILE

operatore	`sizeof` "`sizeof`"
sintassi	`sizeof var`
n. operandi	1
utilizzo	ritorna la dimensione in byte di `var`
associatività	⟸
commutatività	NO

L'operatore `sizeof` viene valutato all'atto della compilazione del programma e diventa un intero costante nel codice generato. Restituisce la dimensione del tipo o del dato alla sua destra, come in `sizeof int`. Per chiarezza, è consuetudine mettere l'operando di `sizeof` tra parentesi e pensare a `sizeof` come a una funzione Sintatticamente, però, tali parentesi sono le parentesi aritmetiche per alterare la priorità degli operatori, non una vera chiamata a funzione.

L'operatore `sizeof` viene usato spesso per l'allocazione dinamica della memoria, quando si vogliano allocare dei vettori di elementi. Nell'esempio seguente, viene allocato un vettore di 10 elementi, ciascuno di tipo `struct buf`, e l'indirizzo dell'area allocata viene assegnato al puntatore `buffers`. Il valore passato a `malloc` è `10 * sizeof(struct buf)`, in quanto la `malloc` si attende il *numero di byte* da allocare. Tale espediente consente di effettuare una allocazione di memoria senza conoscere esplicitamente la dimensione del singolo elemento del vettore: anche se l'insieme dei campi della struttura `struct buf` cambiasse nel tempo, l'istruzione di allocazione resterebbe così com'è.

```
struct buf *buffers = malloc(10 * sizeof(struct buf));
```

L'operatore `sizeof` è utile anche per determinare la dimensione in bit dell'architettura sulla quale si sta compilando il programma, per eseguire operazioni diverse a seconda della dimensione dell'intero. Un esempio è il seguente:

```
if (sizeof(int) == 2) {
  /* codice per processore a 16 bit */
} else if (sizeof(int) == 8) {
  /* codice per 64 bit */
} else {
  /* codice per processore a 32 bit */
}
```

Quando l'operando è il nome di un tipo di dato e non un valore, una variabile o un'espressione, l'uso delle parentesi è obbligatorio (vedi Sezione 7.3).

L'operatore `sizeof` accetta in ingresso anche una espressione, non soltanto variabili (che sono una forma banale di espressioni) o tipi di dato. Nel caso in cui l'argomento dell'operatore sia una espressione, tale espressione NON viene valutata, ma viene restituita soltanto la dimensione del tipo restituito dall'espressione. Questo significa che, per esempio, le istruzioni:

```
  dim = sizeof(i++);
  dim = sizeof(a=4);
```

NON MODIFICANO il valore delle variabili coinvolte nelle espressioni, rispettivamente `i` ed `a`, ma semplicemente a `dim` verrà assegnata la dimensione del valore ritornato dalle espressioni. Quindi, se `i` ed `a` sono per esempio di tipo `int`, le istruzioni precedenti sono equivalenti a

```
  dim = sizeof(int);
```

9.18 MOLTIPLICAZIONE E DIVISIONE

operatore	* asterisco
sintassi	`espr1 * espr2`
n. operandi	2
utilizzo	moltiplica `espr1` per `espr2`
associatività	$\Longrightarrow$
commutatività	SI

L'asterisco in questo uso non provoca ambiguità con la dereferenziazione di puntatore, perché la moltiplicazione ha due operandi e comunque non può essere effettuata sui puntatori.

La moltiplicazione intera non gestisce l'overflow dei valori. Nel seguente esempio, si ha un overflow se il valore di `sec` è minore di -2 o maggiore di 2.

```
int nsec = sec * 1000000000;
```

operatore	/ slash
sintassi	`espr1 / espr2`
n. operandi	2
utilizzo	divide `espr1` per `espr2`
associatività	$\Longrightarrow$
commutatività	NO

La divisione intera scarta il resto.

Nel seguente esempio, vengono utilizzate le operazioni di moltiplicazione e divisione per convertire un valore intero di temperatura da gradi Celsius a Farhenheit:

```
int fahr, cels;
fahr = cels * 9 / 5;
```

L'istruzione seguente effettua la conversione inversa:

```
/* errato per perdita del resto: usare "* 5 / 9" */
cels = fahr / 9 * 5;
```

In questo caso, viene effettuata prima la divisione e poi la moltiplicazione. A causa del troncamento del resto causato dalla divisione, tale istruzione può causare errori inaspettati dovuti alla perdita di precisione. Se `fahr` assume per esempio valori da 0 a 8, la divisione per 9 restituisce 0, causando un'inaccettabile perdita di precisione. È meglio quindi effettuare prima la moltiplicazione, per evitare questo problema:

```
cels = fahr * 5 / 9;
```

Bisogna in questo caso fare attenzione che un valore troppo elevato di `fahr` non causi overflow, ovvero che il risultato della moltiplicazione di `fahr` per 5 non ecceda il valore massimo memorizzabile nel registro che contiene il risultato parziale dell'espressione.

9.19 RESTO DI DIVISIONE INTERA

operatore	% percentuale
sintassi	`espr1 % espr2`
n. operandi	2
utilizzo	restituisce il resto della divisione intera tra `espr1` e `espr2`
associatività	$\Longrightarrow$
commutatività	NO

Esempio:

```
void stampatempo(int s)
{
  int h, m;

  m = s / 60; s = s % 60;
  h = m / 60; m = m % 60;
  printf("%i:%02i:%02i\n", h, m, s);
}
```

9.20 SOMMA E SOTTRAZIONE

operatore	$+$ più
sintassi	`espr1` $+$ `espr2`
n. operandi	2
utilizzo	restituisce la somma tra `espr1` e `espr2`
associatività	$\Longrightarrow$
commutatività	SI’

operatore	$-$ trattino alto
sintassi	`espr1` $-$ `espr2`
n. operandi	2
utilizzo	restituisce la differenza tra `espr1` e `espr2`
associatività	$\Longrightarrow$
commutatività	NO

Uno dei due operandi può essere un puntatore, in tal caso il risultato è un puntatore. Se entrambi gli operandi sono puntatori, il risultato è un numero intero corrispondente alla differenza tra i due indirizzi. Come per la moltiplicazione, non c’è nessun controllo sull’overflow.

```
int a = 200, b = 300;
unsigned int c;

c = a - b; /* un numero positivo: 2 alla 32 meno 100 */
```

9.21 SPOSTAMENTO DEI BIT (SHIFT)

operatore	<< minore minore
sintassi	`var << n`
n. operandi	2
utilizzo	effettua lo scorrimento a sinistra di `n` posizioni dei bit di `var`
associatività	$\Longrightarrow$
commutatività	NO

operatore	>> maggiore maggiore
sintassi	`var >> n`
n. operandi	2
utilizzo	effettua lo scorrimento a destra di `n` posizioni dei bit di `var`
associatività	$\Longrightarrow$
commutatività	NO

Lo shift verso sinistra comporta l'inserimento sulla destra di tanti zeri quante sono le cifre spostate. Lo shift verso destra comporta l'inserimento sulla sinistra di tanti zeri quante sono le cifre spostate.

Per esempio

```
char c, c1;

c = 0x0d;      /* 00001101 */
c1 = c << 1;   /* 00011010 */
c1 = c << 3;   /* 01101000 */
c1 = c << 7;   /* 10000000 */

c = 0xd0;      /* 11010000 */
c1 = c >> 1;   /* 01101000 */
c1 = c >> 3;   /* 00011010 */
c1 = c >> 7;   /* 00000001 */
```

Si noti che lo shift equivale ad una moltiplicazione o divisione per potenze di due, a seconda che si tratti di shift a sinistra o a destra.

Un esempio un pò più complesso è il seguente, il quale effettua la conversione di un pixel dalla codifica rgb888 a rgb565.

```
unsigned char rgb[3];
unsigned short pixel;

pixel = ((rgb[0] >> 3) << 11) +
```

```
((rgb[1] >> 2) << 5) +
(rgb[2] >> 3);
```

Si rimanda all'Approfondimento 2 per maggiori informazioni riguardanti la codicia dei colori in formato RGB.

> La codifica RGB (da Red–Green–Blue, ovvero Rosso–Verde–Blu) viene utilizzata per memorizzare il colore di un punto, e utilizza un certo numero di bit per memorizzare le tre componenti del colore (rosso, verde e blu) che miscelate producono tutti i possibili colori.
> La codifica rgb888 utilizza 8 bit per ciascun colore, mentre la codifica rgb565 utilizza 5 bit per il rosso, 6 bit per il verde e 5 bit per il blu. Per questo il pixel codificato in rgb888 è memorizzato in un array di 3 caratteri (`unsigned char rgb[3]`), mentre il pixel codificato in rgb565 viene memorizzato nello `short pixel` di 2 soli byte.
> Dopo la conversione, bisogna fare attenzione a come si salva `pixel` (16 bit) su macchine big-endian/little-endian . Si veda l'Approfondimento 1 per maggiori dettagli riguardanti l'endianness.

Approfondimento 2: Codifica dei colori in formato RGB.

9.22 CONFRONTO

operatore	> >= maggiore (maggiore uguale)
sintassi	`espr1 > espr2` `espr1 >= espr2`
n. operandi	2
utilizzo	ritorna 1 nel caso in cui `espr1` e' maggiore (maggiore uguale) a `espr2`, altrimenti ritorna 0
associatività	$\Longrightarrow$
commutatività	NO

operatore	< <= minore (minore uguale)
sintassi	`espr1 < espr2` `espr1 <= espr2`
n. operandi	2
utilizzo	ritorna 1 nel caso in cui `espr1` è minore (minore uguale) a `espr2`, altrimenti ritorna 0
associatività	$\Longrightarrow$
commutatività	NO

Esempio

```
int fuorimisura = i > 100 || i < 50;

if (fuorimisura) {
  /* gestione errore */
}

/* conversione da cm in decimi di pollice */
i = i * 1000 / 254;
```

9.23 CONFRONTO: UGUAGLIANZA E DIVERSITÀ

operatore	== uguale uguale
sintassi	`espr1` == `espr2`
n. operandi	2
utilizzo	ritorna 1 se `espr1` è uguale a `espr2`, altrimenti 0
associatività	$\Longrightarrow$
commutatività	SI’

operatore	! = punto esclamativo – uguale
sintassi	`espr1` ! = `espr2`
n. operandi	2
utilizzo	ritorna 1 se `espr1` è diverso da `espr2`, altrimenti 0
associatività	$\Longrightarrow$
commutatività	SI’

Gli operatori di confronto servono a verificare uguaglianza o diversità tra due variabili.

Come sopra, il risultato è 0 oppure 1 in caso di uguaglianza o diversità.

Nell’esempio

```
if (value == 10) { /* istruzioni */ }
```

il blocco di istruzioni viene eseguito se il valore della variabile `value` vale 10, mentre nel seguente

```
if (value != 100) { /* istruzioni */ }
```

il blocco di istruzioni viene eseguito se il valore della variabile `value` è diverso da 100.

L'uso di == viene spesso confuso con l'uso di =: il primo operatore effettua un test e quindi non cambia il valore della variabile, mentre il secondo un assegnamento, cambiando il valore della variabile. Il compilatore non segnala alcun avvertimento a riguardo.

9.24 AND BIT-A-BIT

operatore	& "e" commerciale
sintassi	`espr1` & `espr2`
n. operandi	2
utilizzo	il bit i-esimo del risultato corrisponde all'AND tra il bit i-esimo di `espr1` e quello di `espr2`
associatività	$\Longrightarrow$
commutatività	SI'

L'AND tra due valori binari vale 1 soltanto se entrambi i valori sono 1.
Esempio:

```
int lowbyte = val & 0xff;
int highbyte = val & 0xff00;
```

L'operazione di AND bit-a-bit si dice talvolta *mascheratura*, e viene detta *maschera* la sequenza di bit che viene "applicata" al dato. La mascheratura viene utilizzata per estrarre il valore dei bit nel dato di partenza che hanno posizioni corrispondenti alle posizioni degli "uno" nella maschera. Nell'esempio seguente, il risultato è ottenuto come and bit-a-bit tra il dato e la maschera, ovvero `risultato = dato & maschera`:

```
dato:       0100101001111001
maschera:   0000111100000000
risultato:  0000101000000000
```

L'effetto dell'operazione è quella di avere i bit di `risultato` che sono tutti zero dove i bit della maschera valgono zero, mentre sono uguali ai bit di `dato` nelle posizioni della maschera in cui i bit valgono uno.

Un modo semplice per verificare se un bit vale 1 in un dato è quindi il seguente:

```
int dato, pos;
/* assegnamento di 'dato' ... */
if (dato & (0x01 << pos)) {
  /* elaborazione */
}
```

nel quale l'elaborazione viene effettuata soltanto se il bit di `dato` che si trova in posizione `pos` vale 1. Si noti che lo shift sposta un bit che vale 1 a sinistra di `pos` posizioni, prima di effettuare la mascheratura.

9.25 XOR BIT-A-BIT

operatore	^ accento circonflesso
sintassi	`espr1 ^ espr2`
n. operandi	2
utilizzo	il bit i-esimo del risultato corrisponde allo XOR tra il bit i-esimo di `espr1` e quello di `espr2`
associatività	$\Longrightarrow$
commutatività	SI'

Il risultato dello XOR tra due valori binari vale 1 se i valori sono diversi.

```
while ( /* condizione */ ) {
   /* calcolo */
   led = led ^ 1; /* inversione del bit piu' basso */
}
```

Se `led` vale 11010001, dopo l'operazione diventa 11010000 e viceversa.

9.26 OR BIT-A-BIT

operatore	| barra verticale
sintassi	`espr1 | espr2`
n. operandi	2
utilizzo	il bit i-esimo del risultato corrisponde all'OR tra il bit i-esimo di `espr1` e quello di `espr2`
associatività	$\Longrightarrow$
commutatività	SI'

Il risultato dell'OR tra due valori binari vale 1 se almeno uno dei due valori è 1.

```
flags = flags | FLAG_BUSY;
/* ... */
flags = flags & ~FLAG_BUSY;
```

Anche in questo caso, come già visto per l'AND bit-a-bit, si parla di *mascheratura*.

La prima istruzione imposta a 1 (set) i bit che sono a 1 in `FLAG_BUSY`, mentre la seconda imposta a 0 (reset) i bit che sono a 1 in `FLAG_BUSY`.

Un esempio di mascheratura è il seguente

```
#define FLAG_BUSY (0xd3)    /* 11010011 */
```

```
flags = 0x31;                /* 00110001 */

flags = flags | FLAG_BUSY;   /* 11110011 */
/* ... */
flags = flags & ~FLAG_BUSY;  /* 00100000 */
```

9.27 AND LOGICO

operatore	&& “e” commerciale – “e” commerciale
sintassi	`espr1` && `espr2`
n. operandi	2
utilizzo	il risultato vale 1 solo se entrambi gli operandi sono veri (cioè diversi da zero), altrimenti il risultato è zero
associatività	$\Longrightarrow$
commutatività	SI’/NO

Il secondo operando viene valutato solo se il primo è vero; se il primo è falso il risultato è già noto, quindi il secondo operando non viene valutato.

Il programma completo è contenuto nel file `and.c`.

```
struct t_str;
struct t_methods {
  int (* print)(struct t_str *str);
};
struct t_str {
  int id;
  struct t_methods *methods;
};
/* ... */
  if (strptr && strptr->methods && strptr->methods->print)
    strptr->methods->print(strptr);
/* ... */
```

- il puntatore a funzione `print` è contenuto nella struttura `methods`
- a sua volta `methods` è contenuta nella struttura `strptr`

Il metodo `print` viene chiamato solo se nessun puntatore in gioco è nullo:

1. si controlla che `strptr` non sia nullo;
2. si controlla che `methods` non sia nullo;
3. si controlla che `print` non sia nullo.

L’AND vale 1 solo se TUTTI i puntatori sono diversi da 0 (NULL); in tal caso è possibile eseguire la funzione puntata dal puntatore `print`.

9.28 OR LOGICO

operatore	|| barra verticale – barra verticale
sintassi	`espr1` || `espr2`
n. operandi	2
utilizzo	il risultato vale 1 se almeno uno degli operandi sono veri (cioè diversi da zero), altrimenti il risultato è zero
associatività	$\Longrightarrow$
commutatività	SI'/NO

Se il primo operando è diverso da zero, il secondo operando non viene valutato.

Una nota sulla commutatività: dal punto di vista logico, invertire gli operandi non causa nessuna variazione sul risultato logico dell'operazione. In particolari casi di utilizzo dell'operatore, però, l'ordine può avere rilevanza, come nel caso seguente:

Il programma completo è contenuto nel file `or.c`.

```
if (v[i] || fill_item(&v[i], i) || set_default(&v[i])) {
  /* lavoro sulla struttura puntata da v[i] */
}
```

dove `v[i]` è l'i-esimo elemento di un vettore di puntatori. Il puntatore viene utilizzato all'interno del blocco condizionale, ma deve essere non-nullo perché lo si possa utilizzare correttamente. Così l'istruzione condizionale serve per assicurarsi che almeno una delle tre condizioni si verifichi per l'inizializzazione del puntatore i-esimo. E l'ordine di valutazione è importante in quanto

1. prima si controlla che il puntatore non sia già non-nullo, perché inizializzato precedentemente;
2. se `v[i]` è nullo, si chiama `fill_item`, che *eventualmente* inizializza il puntatore
3. se `fill_item` non inizializza il puntatore, allora gli si assegna un valore di default con `set_default`.

Sia a `fill_item` che a `set_default` è richiesto di ritornare un valore non nullo in caso di inizializzazione effettuata (basta ritornare il valore di v[i]).

Si nota subito che eseguire la valutazione delle espressioni poste in OR tra loro, cambia la logica di funzionamento del programma. Perciò in questo caso l'operazione di OR non si può considerare commutativa.

La priorità di OR è minore di quella di AND, perché OR è assimilabile ad una somma, mentre AND è assimilabile ad una moltiplicazione.

9.29 ESPRESSIONE CONDIZIONALE

L'espressione condizionale è realizzata per mezzo dell'operatore ternario che utilizza il punto di domanda e i due punti

operatore	? : punto di domanda – due punti
sintassi	`espr1 ? espr2 : espr3`
n. operandi	3
utilizzo	se `espr1` è vera, allora `espr2` viene valutata come risultato, altrimenti viene valutata `espr3`
associatività	$\Longleftarrow$
commutatività	NO

È l'unico operatore del C che richiede tre operandi. Il tipo della seconda e della terza espressione deve essere compatibile.

```
printf("%i byte%s in %i file%s", bytes,
       bytes==1 ? "" : "s",
       files, files==1 ? "" : "s");
```

L'istruzione precedente stampa il numero di byte totali (`bytes`) dei file considerati il numero totale di byte (`files`). Le istruzioni condizionali stampano la 's' per il plurale se `bytes` o `files` sono diversi da 1, altrimenti le parole nella stringa di output vengono lasciate al singolare.

Le istruzioni che seguono assegnano il massimo e il minimo di due valori `a` e `b` alle due variabili `max` e `min` rispettivamente, utilizzando l'operatore ternario per il confronto e la selezione del valore da assegnare.

```
max = (a > b) ? a : b;
min = (a < b) ? a : b;
```

9.30 ASSEGNAMENTO

operatore	$=$ uguale
sintassi	`espr1` $=$ `espr2`
n. operandi	2
utilizzo	assegna il valore di `espr2` ad `espr1`; il cui risultato dell'espressione è uguale al valore di `espr2`
associatività	$\Longrightarrow$
commutatività	NO

L'operando di sinistra deve essere una variabile o una struttura dati o una espressione equivalente. Tale operando si chiama *lvalue*, abbreviazione di "left value". I messaggi di errore del compilatore relativi ad lvalue si riferiscono ad assegnamenti erronei. Il valore di una espressione di assegnamento ha come valore il valore assegnato.

```c
/* a = (b = (c = 0)) */
a = b = c = 0;

/* sintatticamente valido, equivalente a if(0) */ ;
if (i = 0)

/* bene */
stat_array[12]->st_mode = 0;

/* lvalue non valida: un vettore non e' assegnabile */
"nome" = s;

/* lvalue non valida: come sopra ma piu' evidente */
3 = i;
```

Nell'esempio

```c
int value = index == 10;
```

sono combinati i due operatori di assegnamento e confronto. Il risultato dell'istruzione è quello di assegnare il valore 1 alla variabile `value` se la variabile `index` vale 10, mentre assegna il valore 0 se `index` è diverso da 10. In pratica viene prima valutata l'espressione `index == 10`, e viene assegnato il valore 0 piuttosto che 1 a seconda che l'espressione sia rispettivamente falsa o vera.

L'uso di == viene spesso confuso con l'uso di =: il primo operatore effettua un test e quindi non cambia il valore della variabile, mentre il secondo un assegnamento, cambiando il valore della variabile. Il compilatore non segnala alcun avvertimento a riguardo. Vedi il secondo esempio.

9.31 FORME ABBREVIATE DI ASSEGNAMENTO

Dal momento che alcune forme di assegnamento sono molto comuni, esistono delle forme abbreviate per l'assegnamento, realizzate dai seguenti operatori:

operatore	*= /= %= += -= <<= >>= &= ^= |= `op uguale` (dove `op` vale: asterisco – slash – percentuale – più – trattino alto – doppio minore – doppio maggiore – "e" commerciale – accento circonflesso – barra verticale)
sintassi	`espr1 op= espr2`
n. operandi	2
utilizzo	sono forme concise di `expr1 = expr1 op expr2`
associatività	⟸
commutatività	NO

Esempi

```
m = s / 60; s %= 60; /* da secondi a minuti e secondi */
flags |= FLAG_BUSY;  /* alzo un bit */
flags &= ~FLAG_BUSY; /* abbasso un bit */
```

La seconda istruzione della prima riga equivale a `s = s % 60`, mentre le altre due sono equivalenti rispettivamente a `flags = flags | FLAG_BUSY` e `flags = flags & FLAG_BUSY`.

Infine, le tre istruzioni seguenti sono equivalenti:

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

9.32 OPERATORE VIRGOLA

operatore	, virgola
sintassi	`espr1 , espr2`
n. operandi	2
utilizzo	valuta l'espressione `espr1` ignorandone il risultato, poi valuta l'espressione `espr2` che vale come risultato dell'operazione "virgola"
associatività	$\Longrightarrow$
commutatività	NO

È usato principalmente nei cicli `while` e per fare cicli `for` con due o più indici.

```
while(next_number(&i), i) {
  /* il nuovo i e' diverso da zero */
}

for (p = v, i = 0; i<32; p++, i++) {
  /* p scorre il vettore fino a
   * un massimo di 32 elementi
   */
}
```

Capitolo 10

CLASSI DI MEMORIA

In C, lo spazio dei nomi di variabili e funzioni è "piatto", non esiste cioè il supporto per *namespace* separati. Variabili e funzioni non possono avere lo stesso nome, perché ad un nome può essere associato un solo indirizzo, sia esso codice o dati.

10.1 LA VISIBILITÀ DI DATI E FUNZIONI

Per *visibilità* o *scope* di una variabile o di una funzione si intende la parte di programma che può accedervi.

Le variabili dichiarate all'interno di una funzione sono dette *locali* o *automatiche*. Le variabili possono anche essere dichiarate al di fuori del corpo delle funzioni (o di un generico blocco di istruzioni), in tal caso sono dette *globali*.

A differenza delle variabili globali, le variabili locali, o *automatiche*, sono visibili solo all'interno del blocco in cui sono dichiarate. Tale blocco può essere una funzione o anche un'istruzione composta racchiusa tra graffe, sia essa il corpo di un costrutto di controllo come `if` o `for`, oppure un'istruzione composta a se stante. Le variabili locali a un blocco sono allocate sullo stack, mentre non è possibile definire "funzioni locali" all'interno di un blocco.

Se una variabile definita all'interno di un blocco ha lo stesso nome di un'altra variable, globale o locale, all'interno del blocco il nome si riferisce alla sua definizione più interna.

Gli argomenti di una funzione sono variabili locali della funzione stessa.

Il seguente esempio mostra alcuni esempi di dichiarazione di variabili locali e globali.

Il programma è contenuto nel file `scope.c`.

```
1  #include <stdio.h>
2
3  int c = 1;
4
5  void func1(int c)
6  {
7    printf("[func1, pre if ] %d\n", c);
8    if (1) {
9      int c = 5;
10     printf("[func1, if     ] %d\n", c);
11   }
12   printf("[func1, post if] %d\n", c);
13 }
14
15 void func2()
16 {
17   printf("[func2, pre if ] %d\n", c);
18   if (1) {
19     int c = 6;
20     printf("[func2, if     ] %d\n", c);
21   }
22   printf("[func2, post if] %d\n", c);
23 }
24
25 int main(int argc, char **argv)
26 {
27   int c = 2;
28
29   printf("[main,  pre if ] %d\n", c);
30   if (1) {
31     int c = 3;
32     printf("[main,  if     ] %d\n", c);
33   }
34   printf("[main,  post if] %d\n", c);
35
36   func1(4);
37   func2();
38
39   return 0;
40 }
```

La variabile `c` è definita globalmente alla linea 3, ma le varie funzioni dichiarano variabili con lo stesso nome con vari livelli di visibilità. Ciascuna funzione, infatti, contiene un blocco condizionale all'interno del quale è dichiarata una variabile `c` che viene quindi utilizzata solo all'interno del blocco stesso (cosa stamperà il programma in corrispondenza delle `printf` che contengono la stringa "post if"?).

Nella funzione `func1`, il parametro `c` viene utilizzato alla linea 7 all'esterno del suo blocco `if`, invece della variabile globale, mentre la funzione `func2` utilizza, sempre all'esterno del suo blocco `if` (linea 17), la variabile globale. La funzione `main`, invece, dichiara localmente una variabile chiamata `c` (linea 27), che viene quindi utilizzata al posto della variabile globale.

Tabella 10.1 Scope e lifetime delle variabili.

Classe	Scope	Lifetime
extern	globale	permanente
auto	locale	temporaneo
static	locale	permanente
register	locale	temporaneo

10.2 CLASSI DI MEMORIZZAZIONE

Le variabili si dividono in quattro classi di memorizzazione, o *storage classes*, a seconda della loro visibilità (scope) e del tempo di vita, il cosiddetto *lifetime*. Il lifetime si riferisce all'intervallo temporale nel quale una variabile, e in particolare il relativo spazio allocato in memoria, può essere acceduta. Se lo spazio in memoria è disponibile per tutta la durata del programma, allora si parla di lifetime permanente, mentre se tale spazio è allocato temporaneamente il lifetime è temporaneo.

Le varie combinazioni sono riportate in Tabella 10.1.

Una variabile ha visibilità globale se può essere utilizzata nell'intero modulo (file) in cui è definita, locale se può essere utilizzata solo all'interno del blocco in cui è definita.

Il tempo di vita è permanente per le variabili allocate staticamente, cioè per le variabili il cui spazio in memoria viene allocato dal compilatore all'inizio del programma e deallocato alla fine. Il tempo di vita è temporaneo per le variabili allocate dinamicamente, tipicamente utilizzando esplicitamente le funzioni per l'allocazione dinamica della memoria come `malloc`, oppure allocate automaticamente dal compilatore come nel caso di variabili dichiarate all'interno di funzioni o altri blocchi di codice, che vengono rilasciate al termine dell'esecuzione del blocco in cui sono definite.

Il programma è contenuto nel file `static.c`.

```
1  #include <stdio.h>
2
3  int c1 = 1;
4
5  void func() {
6    int c3 = 3;
7    static int c4 = 1;
8
9    printf("func: c1 %2d   c3 %2d   c4 %2d\n", c1, c3, c4);
10   c4++;
11   c3++;
12 }
13
14 int main(int argc, char **argv)
15 {
16   int i;
17
18   //printf("main: c3 %d\n", c3);
19
20   for (i = 0; i < 5; i++) {
21     c1 = c1 + 2;
22     func();
```

```
23      }
24
25      return 0;
26  }
```

La variabile `c1` (linea 3) è globale, allocata staticamente, e può essere utilizzata in tutto il file sorgente. La variabile `c3` (linea 6) è allocata dinamicamente ogni volta che la funzione `func` viene richiamata e può essere utilizzata soltanto all'interno della funzione stessa. Per questo motivo se si tenta di utilizzare `c3` nel `main` si ha un errore in compilazione alla linea 18. Per quanto riguarda la variabile `c4`, questa è dichiarata all'interno della funzione `func`; la variabile può essere usata soltanto all'interno di tale funzione, ma essendo dichiarata `static`, essa è allocata staticamente, e quindi l'area di memoria ad essa associata non viene rilasciata quando la funzione termina. Questo permette a tale variabile di conservare il proprio valore da una chiamata all'altra della funzione. Sia `c3` che `c4` vengono incrementate ad ogni chiamata della funzione `func`, ma si comportano in modo molto diverso a causa del differente lifetime.

Un esempio di esecuzione del programma è il seguente:

```
main: c1 1
func: c1 1 c3 3 c4 1
func: c1 1 c3 3 c4 2
func: c1 1 c3 3 c4 3
func: c1 1 c3 3 c4 4
func: c1 1 c3 3 c4 5
```

Si noti in particolare come il valore di `c4` mantenga il proprio valore (incrementato di volta in volta) ad ogni chiamata della funzione `func`. Questo è un semplice modo per tenere traccia di quante volte viene richiamata una funzione.

10.3 ALLOCAZIONE DI MEMORIA

Il linguaggio C non offre primitive di gestione di memoria[20] e nemmeno la cosiddetta "raccolta della spazzatura", la *garbage collection* che viene messa a disposizione da altri linguaggi come Java.

La memoria usata dai programmi può essere di tre tipi:

1. statica, che è dichiarata in compilazione, cui il linker assegna un indirizzo immutabile
2. dinamica, cioè allocata durante il funzionamento del programma, per esempio chiamando `malloc` e accedendo allo spazio così ottenuto tramite un puntatore
3. automatica, che viene allocata sullo stack e scompare al termine del blocco di codice che la dichiara

Una variabile statica è inizializzata a zero, a meno che il programma non dichiari un valore costante da precaricare nella variabile. Le variabili inizializzate sono salvate su disco e risiedono nel "segmento dati" del programma e del file eseguibile ELF; le variabili non inizializzate esplicitamente nel programma stanno nel "segmento bss" del programma, una zona di memoria che viene allocata e azzerata prima dell'esecuzione del programma, il file su disco non contiene una copia del bss ma solo la sua dichiarazione.

Una variabile dinamica risiede in un'area di memoria che viene richiesta al sistema durante il funzionamento del programma. Al momento dell'allocazione non si possono fare assunzioni sul

[20]Come new, creatori e distruttori tipici dei linguaggi ad oggetti.

contenuto di tale memoria: potrebbe essere azzerata ma potrebbe contenere informazioni residue di precedenti allocazioni poi liberate. Ad ogni `malloc` deve corrispondere una `free`, in mancanza della quale abbiamo una situazione di perdita di memoria (*memory leakage*) e l'occupazione di memoria del programma in esecuzione aumenterà in continuazione. Capita spesso, infatti, che in un programma si succedano frequenti chiamate a molte `malloc` e `free`. Se per errore non si libera la memoria precedentemente allocata e non più utilizzata, questa rimane occupata e inutilizzabile dal sistema (non soltanto dal nostro programma). Se ciò continua ad accadere per tutto il tempo in cui il programma rimane in esecuzione, si comprende come la memoria allocata al programma tenda ad aumentare senza che però venga utilizzata proficuamente.

Una variabile automatica è una variabile locale di una funzione o di un blocco di codice, risiede sullo stack e non viene inizializzata a meno che il programmatore non imponga un valore a tale variabile; in tal caso il codice macchina generato dal compilatore contiene le istruzioni necessarie a riempire la variabile come richiesto. La memoria delle variabili automatiche, essendo parte dello stack del programma, non è più utilizzabile al termine della procedura che definisce la variabile stessa.

Alcuni esempi di dichiarazione di variabili sono i seguenti:

```
1  int i;
2  int v[4] = {2,1};
3  int j = f(3, i);
4
5  int *f(int x, int y)
6  {
7    int z;
8    int a = 0, b = 1, c = 2;
9    int *p = malloc(4 * sizeof(int));
10   int *q, *r = &z;
11
12   *q = y;
13   *r = y;
14   if (x) return p;
15   else return &z;
16 }
```

Le linee di codice dalla 1 alla 3 dichiarano variabili globali, mentre le linee dalla 7 alla 10 effettuano la dichiarazione di variabili locali. Infine, le istruzioni dalla 12 alla 15 sono degli assegnamenti.

- la dichiarazione di `i` effettuata alla linea 1 fa in modo che il dato, memorizzato nel segmento bss, sia inizializzato a zero;

- il vettore `v` viene inizializzato esplicitamente alla linea 2; il codice specifica i valori da assegnare ai primi due elementi del vettore, mentre gli altri sono inizializzati a zero; dopo l'inizializzazione, si avrà `v = {2,1,0,0}`; la variabile è memorizzata nel segmento dati;

- l'inizializzazione della variabile globale `j` alla linea 3 rappresenta un errore: il valore da usare per l'inizializzazione non è noto in fase compilazione e quindi non può essere utilizzato per inizializzare la variabile globale;

- la dichiarazione della variabile `z` alla linea 7 è corretta, ma trattandosi di una variabile automatica (locale alla funzione), il suo valore non è noto fintanto che non viene fatto un assegnamento esplicito;

- tutte le variabili dichiarate alla linea 8 sono allocate al momento dell'esecuzione della funzione; l'inizializzazione è quindi effettuata a run-time, una volta allocata la memoria necessaria;

- l'inizializzazione del puntatore `p` alla linea 9 viene fatta correttamente a run-time: dopo l'allocazione della variabile locale viene chiamata la `malloc` la quale ritorna il valore con il quale inizializzare la variabile;
- alla linea 10 sono dichiarati due puntatori, a uno dei quali, `r`, viene correttamente assegnato il valore dell'indirizzo di `z`
- l'assegnamento della linea 12 è un errore: il puntatore `q` non è stato assegnato, e quindi contiene un indirizzo non noto a priori; in pratica, con questa istruzione si sta tentando di scrivere il valore di `y` in una locazione ignota della memoria; questo provoca tipicamente un errore di *segmentation fault* oppure la corruzione della memoria utilizzata dal programma in esecuzione;
- l'assegnamento alla linea 13 è invece corretto: `r` contiene l'indirizzo di `z`; in pratica, viene assegnato il valore della variabile `y` alla variabile `z`;
- l'istruzione alla linea 14 è corretta, in quanto la funzione ritorna un puntatore (`p`) ad un'area di memoria allocata all'interno della funzione stessa, che rimane disponibile anche una volta che la funzione termina;
- l'instruzione alla linea *??* è invece errata, in quanto ritorna l'indirizzo della variabile `z` ma, essendo `z` una variabile temporanea il cui spazio allocato in memoria viene rilasciato al termine della funzione, il valore ritornato dalla funzione stessa corrisponderà all'indirizzo di memoria non più allocata.

10.4 VARIABILI AUTO

Tutti gli esempi visti trattavano variabili definite, implicitamente, `auto`. Appartengono a questa classe le variabili locali ad un blocco, le quali vengono allocate durante l'esecuzione del blocco nel quale sono dichiarate e rilasciate al termine dello stesso, non possono perciò essere utilizzate all'esterno del blocco di definizione.

Nel seguente esempio

```
funzione()
{
  auto int x = 10;
  auto int y;
}
```

Nel quale la variabile `x` viene inizializzata, mentre `y` contiene un valore casuale in quanto non esplicitamente inizializzata.

La specificazione con la parola chiave `auto` può generalmente essere omessa: di fatto il programma poteva essere scritto in modo equivalente come segue:

```
funzione()
{
  int x, y;
  x = 10;
}
```

Si tenga presente che array e strutture locali non possono essere inizializzati.

10.5 VARIABILI `REGISTER`

La classe `register` avverte il compilatore che le variabili associate dovrebbero essere memorizzate in registri della CPU. A causa delle risorse limitate tale richiesta non viene necessariamente rispettata.

Fondamentalmente, l'uso della classe `register` è un tentativo di migliorare la velocità di esecuzione, ovvero è una cosiddetta "direttiva di ottimizzazione". In genere vengono definite `register` le variabili di un ciclo.

```
{
  register int i;
  for (i = 0; i < 10; i++) {
    ....
  }
}
```

Non è possibile ottenere l'indirizzo di una variabile `register`, il compilatore perciò segnalerà un errore nell'esempio seguente:

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

NOTA La direttiva `register` è da considerare obsoleta, in quanto il compilatore è in grado di determinare automaticamente se una variabile è bene che sia memorizzata in un registro o meno.

10.6 VARIABILI STATIC

La parola chiave *static* è un qualificatore per codice e dati, che serve per cambiare le regole di visibilità.

Un simbolo globale, sia esso una funzione o una variabile, se dichiarato `static` non è visibile all'esterno del file ove è definito, nel caso di programmi costituiti da più file sorgenti (i quali sono trattati in dettagli nel capitolo 13), perché il suo nome non viene reso disponibile al linker. Una variabile locale, se `static`, viene allocata nello spazio dati globale, ma senza esportarne il nome; permette quindi di avere uno stato persistente tra le varie invocazioni del blocco in cui è definita. Per esempio:

```
int i;          /* globale */
static int j;   /* globale, ma visibile solo in questo file */

static int invert(int i) /* invert puo' essere chiamata solo in questo file */
{
  int j;   /* allocata sullo stack */
  j = -i;  /* variabili locali, "i" e' l'argomento della funzione  */
  return j;
}

int count(void)  /* count e' definita globalmente nel programma */
{
  static int i;  /* locale ma persistente, inizializzata a zero */
  return ++i;    /* incrementa il contatore e ritorna il valore */
}
```

Il programma precedente mostra una serie di dichiarazioni di variabili statiche/dinamiche, locali/-globali. L'esempio è abbondantemente commentato.

Appartengono quindi alla classe `static` le variabili locali ad un blocco, allocate però staticamente. A differenza delle variabili `auto`, il valore delle variabili `static` si conserva da una chiamata all'altra della stessa funzione. In altri termini il lifetime di una variabile `static` è permanente. Una variabile `static` mantiene quindi il suo valore, pur rimanendo invisibile al di fuori del blocco di definizione.

```
func()
{
  static int counter = 1;
  printf("Funzione eseguita %d"
    " volte\n", counter++);
}
```

Esempio di uso di un blocco e di variabili statiche per il debugging di un programma:

```
{
  static int cont = 0;
  fprintf(stderr, "** debug: cont=%d v=%d\n",
    cont++, v);
}
```

Questo blocco di codice, posto in opportuni punti del programma, permette di tenere traccia di quante volte viene eseguito, e permette quindi di capire quante volte il flusso di esecuzione del programma “passa” da questo punto. Il tutto viene gestito mediante una variabile locale al blocco, senza quindi che vi sia il rischio di modificare erroneamente tale variabile in altri punti del programma.

10.7 VARIABILI EXTERN

Vengono poste automaticamente nella classe di memorizzazione `extern` le variabili dichiarate a livello globale, cioè al di fuori di qualsiasi blocco o funzione. Si può accedere al valore di queste variabili anche all’interno di un blocco o di una funzione, a meno che una variabile con il medesimo nome venga dichiarata in quel blocco. In questo caso, non è possibile utilizzare lo stesso identificatore per accedere a due variabili distinte, quindi la variabile globale risulta non indirizzabile.

```
int var_globale;

int main()
{
  extern int var_globale;
  /* ... */
  var_globale = 10;
  /* ... */
  printf("%d", var_globale);
  return 0;
}
```

Se la parola chiave `extern` viene esplicitamente utilizzata si indica al compilatore che la variabile viene definita altrove, cioè si dichiara cioè semplicemente l’esistenza di una variabile di quel tipo. Non viene allocato spazio per la variabile dichiarata `extern`, poiché lo spazio è allocato in corrispondenza della dichiarazione della variabile senza la clausola `extern`.

In genere non si usa la `extern` all’interno di una funzione per indicare che la variabile utilizzata è globale: si usa la variabile e basta, in quanto le variabili globali sono implicitamente disponibili nella funzione.

L’uso della `extern` è invece utile quando, in programmi composti da più file, la definizione di una variabile e/o funzione viene fatta in un file diverso rispetto al file sorgente che la utilizza. In quest’ultimo file sorgente la dichiarazione diviene `extern`.

10.8 INIZIALIZZAZIONI

In assenza di inizializzazioni:

- le variabili globali vengono inizializzate a 0;
- le variabili `static` vengono inizializzate a 0;
- le variabili semplici `static` e globali vengono inizializzate tramite espressioni costanti;
- le variabili locali possono utilizzare anche valori definiti in precedenza o i valori restituiti da una funzione.

Alcuni esempi di inizializzazione sono i seguenti:

```
int a = 1;
char dollaro = '\$';
long y = 10000 * 20000;

long w = y * 2;   /* solo per variabili locali */
```

Vettori e strutture possono essere inizializzati solo se globali. Per esempio:

```
int vettore[4] = { 1, 4, 5, 7 };
int vet1[4] = { 12, 13 };
int mesi[] = {
        31, 28, 31, 30, 31, 30,
        31, 31, 30, 31, 30, 31};
char saluto[4] = {'c', 'i', 'a', 'o'};
char sal1[] = {'c','i','a','o','\0'};
char sal2[] = "ciao";
char sal3[4] = "ciao";
```

Capitolo 11

IL PREPROCESSORE

Il preprocessore è un processore di testi che elabora il contenuto di un file sorgente prima della compilazione vera e propria.

Il preprocessore è un programma che opera sostituzioni tipografiche sul codice sorgente prima che tale codice venga visto dal compilatore vero e proprio.

Anche se il preprocesso è un'operazione formalmente distinta dalla compilazione, il preprocessore fa parte del compilatore e delle specifiche del linguaggio; ogni sorgente C viene preprocessato.

Tutte le righe nel codice sorgente che iniziano con il carattere '#' (*diesis*, *cancelletto*, *hash*) sono direttive per il preprocessore. Tali direttive permettono, tra le altre cose di:

- includere (fisicamente) altri file all'interno del proprio sorgente;
- ridefinire il significato degli identificatori, tramite sostituzione puramente tipografica nel codice sorgente;
- disabilitare condizionalmente parti di codice in fase di compilazione, eliminando fisicamente il testo prima che il compilatore lo veda.

NOTA Come si intuisce, il preprocessore è uno strumento potente ma molto pericoloso; per esempio, il compilatore non può effettuare il controllo degli errori sulle parti di codice disabilitate.

Le modifiche apportate al file sorgente riguardano soprattutto:

- l'eliminazione dei commenti
- l'inclusione dei file

- la sostituzione di costanti simboliche e macro

Il preprocessore viene principalmente usato per includere altri file e definire nomi simbolici per riferirsi a dati numerici. L'inclusione dei file di header serve a poter accedere ai prototipi delle funzioni, alle dichiarazioni delle strutture dati e delle variabili globali definite esternamente al proprio programma (ad es. nelle librerie). Normalmente la documentazione di una funzione di libreria specifica quale header occorre includere per passare al compilatore le informazioni necessarie.

11.1 LA DIRETTIVA #DEFINE

La direttiva #define viene usata per definire delle *macro*. Le macro, che possono avere eventualmente dei parametri, sono utilizzate per sostituire del testo all'interno del programma prima della compilazione.

Dopo una definizione della forma

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

Tutte le successive occorrenze di `nome` che non sono racchiuse tra doppi apici sono sostituite da `testo-da-sostituire`. La stringa di testo `testo-da-sostituire` va dallo spazio dopo nome fino alla fine della linea; può continuare su linee successive se l'ultimo carattere della linea è \, il quale fa ignorare il carattere di a capo al precompilatore. Per esempio, è possibile codificare con delle macro i codici di errore gestiti dal programma:

```
#define ERR_NOERROR    0
#define ERR_INVALID    1
#define ERR_NODATA     2
#define ERR_PERMISSION 3
```

Per una convenzione universalmente accettata, le costanti definite tramite preprocessore si scrivono in maiuscolo come mostrato qui sopra, in modo da essere subito riconoscibili leggendo il testo del programma, per non confonderle con le variabili.

Le macro vengono spesso usate anche per realizzare piccole "pseudo-funzioni", sfruttando la possibilità di passare dei parametri alla macro, come nell'esempio seguente:

```
#define SQUARE(a) a*a
```

Funzioni di questo tipo presentano vantaggi in termini di velocità di esecuzione, in quanto non si realizza una chiamata a funzione al momento dell'esecuzione, ma il codice della macro viene sostituito in fase di compilazione, evitando quindi l'overhead della chiamata.

Purtroppo tale prassi si presta ad errori molto subdoli. Nell'esempio riportato, l'errore si manifesta per esempio in "SQUARE(1+2)" che diventa "1+2*1+2" cioè 5 invece di 9, come ci si attenderebbe. Inoltre, quando un argomento di macro appare più di una volta nell'espansione, la macro non può essere equivalente ad una funzione perché operatori come "++" appaiono ripetuti nel testo effettivo del programma, con effetti non desiderati.

11.2 LA DIRETTIVA #INCLUDE

Il preprocessore sostituisce ogni riga della forma:

```
#include <nome-file>
```

oppure

Tabella 11.1 Alcuni header standard del C.

header	contenuto
<math.h>	funzioni e costanti matematiche
<stdio.h>	gestione dell'I/O
<string.h>	gestione delle stringhe
<stdlib.h>	funzioni standard

```
#include "nome-file"
```

con il contenuto del file "`nome-file`".

Se il file incluso è specificato con le parentesi ad angolo viene cercato tra quelli di sistema, se è specificato con le virgolette viene cercato prima nella directory corrente. Esempio:

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

Quindi, nel primo caso il file da includere `stdio.h` viene ricercato in una o più directory standard, che sui sistemi Unix è `/usr/include`, mentre nel secondo caso il file `myheader.h` viene ricercato prima nella directory corrente e poi nelle directory standard.

Il file incluso può contenere qualunque porzione di codice C, comprese altre direttive `#include`. In genere contiene direttive `#define` e dichiarazioni di variabili e funzioni. Funzioni, tipi, macro della libreria del C sono definiti in alcuni header file standard. Alcuni dei file di intestazione maggiormente utilizzati sono riposrtati in Tabella 11.1.

In generale è buona regola non mettere negli header-file il codice delle funzioni, ma solo la loro dichiarazione. La definizione delle funzioni sarà posta all'interno di file `.c` o nei file oggetto già compilati che verranno opportunamente linkati al programma.

11.3 LA DIRETTIVA #IF E #IFDEF

Si possono introdurre segmenti di codice in dipendenza da particolari condizioni. Il costrutto seguente valuta una espressione intera costante, il cui valore deve essere noto all'atto della compilazione:

```
#if espressione-costante-intera
   /*
    * questo codice viene considerato
    * solo se l'espressione risulta
    * diversa da 0
    */
   /*
    * endif termina la sezione
    * condizionale
    */
#endif
```

Tutte le righe di codice comprese tra `#if` e `#endif` vengono incluse nel file che verrà passato al compilatore solo se l'espressione è diversa da 0.

Similmente, il costrutto

```
#ifdef macro
        /*
         * questo codice viene considerato
```

```
         * solo se "macro" e' gia' stata definita
         */
#endif

#ifndef macro
        /*
         * questo codice viene considerato
         * solo se "macro" non e' stata definita
         */
#endif
```

valuta solo se il simbolo X è definito (nel senso di `#define`) o meno.

In `#if` oltre a numeri, simboli definiti in precedenza e operatori interi è possibile usare la forma "`defined(X)`". Per evitare troppi livelli condizionali e troppi `#endif` si può usare `#elif` con il significato di "else if". Per esempio, il seguente codice

```
#ifdef X
  /* codice 1 */
#else
#ifdef Y
  /* codice 2 */
#else
#ifdef Z
  /* codice 3 */
#endif
#endif
#endif
```

diviene il più leggibile

```
#ifdef X
  /* codice 1 */
#elif Y
  /* codice 2 */
#elif Z
  /* codice 3 */
#endif
```

Un esempio pratico è il seguente

```
#if SYSTEM == MSDOS
        #define HDR "msdos.h"
#elif SYSTEM == SYSV
        /* elif equivale ad un else if */
        #define HDR "sysv.h"
#else
        #define HDR "default.h"
#endif
#include HDR
```

È possibile utilizzare la direttiva `#if` per eliminare porzioni di codice senza cancellarle, per esempio in fase di debugging:

```
#if 0
  /* codice da non considerare */
#endif
```

Una volta eliminati i problemi si può velocemente ripristinare il codice sostituendo 0 con 1

```
#if 1
  /* codice ripristinato */
#endif
```

Capitolo 12

I FILE

In questo capitolo saranno trattati gli approcci e le relative funzioni per l'accesso ai file. Quando si parla di accesso a file si intende la lettura e/o scrittura di dati da e verso file.

12.1 FILE BINARI E FILE DI TESTO

Nella trattazione che segue riguardo all'accesso a file da parte di programmi scritti in C, è importante distinguere tra due tipologie di file: i file binari ed i file di testo. La distinzione è necessaria in quanto si utilizzano in genere funzioni diverse per la corretta interazione con i file a seconda del tipo di file.

I file di testo sono file che contengono soltanto caratteri ASCII stampabili ed alcuni caratteri di controllo che servono a definire la struttura del file, come l'andata-a-capo e il carattere di tabulazione. Questi file sono visualizzabili e modificabili usando i classici editor di testo come `emacs` e `vi`, oppure con comandi come `cat`, `more`, `less`, ecc.

I file binari, al contrario dei file di testo, non sono leggibili correttamente con editor di testo, e non sono gestibili con i comandi elencati sopra. Questo perché i file binari non contengono solo i caratteri ASCII stampabili, come caratteri alfanumerici e segni di interpunzione, ma possono contenere tutti i caratteri ASCII.

È possibile accedere ad un file di testo anche usando le funzioni per l'accesso a file binari, in quanto un file di testo non è altro che un "caso particolare" di file binario che utilizza un sottoinsieme (i caratteri stampabili) dei caratteri ASCII. Quindi è possibile leggerli a blocchi di byte come per i file binari. L'opposto non è sempre agevole, in particolare per la lettura. Infatti, le funzioni di accesso a file di testo si aspettano che il file di ingresso sia strutturato in righe separate dall'andata a capo. Dal momento che un file binario non è strutturato in righe, la sua lettura è complicata e può portare ad errori di vario genere.

Esempi di file di testo sono i file sorgente dei programmi, mentre esempi di file binari sono costituiti dai file eseguibili, dai file compressi (zip, ecc.), da alcuni tipi di file di immagini (jpeg, gif, tiff, ecc.). Tipicamente il tipo di file può essere individuato dalla sua estensione, ovvero dagli uno i più caratteri finali del nome del file, che solitamente segue il carattere "punto". Esempi di estensione sono:

- `.c` per i file sorgente di programmi in C (testo)
- `.zip` per i file compressi in formato zip (binario)
- `.jpg` o `jpeg` per le immagini compresse in formato JPEG (binario)

Per comprendere meglio la differenza tra le due tipologie di file, si pensi di dover memorizzare un numero intero in un file. La sua rappresentazione cambia a seconda che lo si memorizzi in un file di testo piuttosto che in un file binario. Un numero intero memorizzato in un file si presenta come una *stringa di caratteri numerici* se viene memorizzato in un file di testo, e con i caratteri ASCII che ne rappresentano la codifica binaria nel caso di un file binario.

Per esempio, il numero decimale "100000" (centomila), può essere rappresentato in un file di testo dai caratteri

```
'1', '0', '0', '0', '0', '0'
```

mentre in formato binario, a seconda della codifica usata dalla macchina, può essere rappresentato con la sequenza di 4 byte 0x000186a0, dove ciascun byte

```
0x00, 0x01, 0x86, 0xa0
```

non necessariamente corrisponde ad un carattere ASCII stampabile.

12.2 ACCESSO A FILE

L'utilizzo del filesystem, e quindi l'accesso ai file, avviene attraverso una serie di funzioni della libreria standard.

Per un corretto utilizzo delle funzioni è necessario includere il file di intestazione `stdio.h` con l'istruzione:

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

Le funzioni (o macro) di libreria standard per l'utilizzo dei file sono le seguenti:

- `fopen`, `fclose`: apertura e chiusura di file
- `fflush`: forzare la scrittura dei dati
- `fread`, `fwrite`: I/O di dati binari
- `fgets`, `fputs`: I/O di linee
- `fscanf`, `fprintf`: funzioni di I/O formattato

Per l'accesso ad un file, il riferimento al file desiderato viene mantenuto per mezzo di un puntatore di tipo `FILE`, definito in `stdio.h`, come

```
FILE * fp;
```

Tutte le funzioni che effettuano l'I/O da e su file utilizzano tali tipi di puntatori come parametri.
Nello stesso header file sono definite, tre variabili di tipo `FILE`:

- `stdin` fa riferimento allo standard input (tipicamente la tastiera)
- `stdout` fa riferimento allo standard output (tipicamente il video)
- `stderr` fa riferimento allo standard error (tipicamente il video)

che vengono aperti automaticamente dal programma quando questo viene eseguito.

12.3 APERTURA E CHIUSURA DI FILE

Per poter utilizzare un file, questo deve essere *aperto*. Per l'apertura di un file si utilizza la funzione `fopen`, descritta nella Sezione 12.3.1.
Una volta che il file è stato scritto e/o letto, il file deve essere "chiuso" utilizzando la funzione `fclose` descritta nella Sezione 12.3.2.

12.3.1 Apertura di file

Le variabili di tipo `FILE` sono inizializzate chiamando la funzione di libreria `fopen`, la quale è dichiarata in `stdio.h` come segue:

```
FILE *fopen(char *path, char *mode);
```

Come si nota, essa accetta due parametri di tipo stringa (puntatore a carattere):

- `path` contiene il percorso e il nome del file da aprire;
- `mode` specifica il *modo* con il quale aprire il file.

Un file può infatti essere aperto per diversi scopi: lettura, scrittura e append, essendo quest'ultima una forma particolare di scrittura. La stringa `mode` può quindi essere una delle seguenti:

- r apre il file in lettura; l'accesso al file avviene dal suo inizio;
- r+ apre il file in lettura e scrittura; l'accesso al file avviene dal suo inizio;
- w apre il file in scrittura; se il file esiste, esso viene sovrascritto, altrimenti un nuovo file viene creato; l'accesso al file avviene dal suo inizio;
- w+ apre il file in lettura e scrittura; se il file esiste, esso viene sovrascritto, altrimenti un nuovo file viene creato; l'accesso al file avviene dal suo inizio;
- a apre il file in append, cioè per una scrittura che avviene a partire dalla fine del file; se il file non esiste, esso viene creato;
- a+ apre il file in lettura e append; se il file non esiste, un nuovo file viene creato; l'accesso in lettura avviene dall'inizio del file, mentre la scrittura è sempre effettuata alla fine del file.

Per completezza, c'è da precisare che è anche possibile aggiungere il carattere "`b`" alla fine o in mezzo a qualsiasi precedente combinazione di caratteri (es., ottenendo `wb` o `ab+`). Questo carattere indica esplicitamente che il file da aprire è binario. In genere questa indicazione esplicita è necessaria per ragioni di portabilità, soltanto se si prevede di usare il programma su sistemi operativi eterogenei. Molti sistemi, infatti, trattano i file binari e i file di testo allo stesso modo, quindi il fatto di specificare

la b risulta superfluo, ma altri sistemi operativi potrebbero operare delle distinzioni, e quindi potrebbe essere necessario specificare esplicitamente il tipo del file trattato.

La funzione fopen restituisce un puntatore di tipo FILE correttamente inizializzato se l'operazione ha avuto successo, altrimenti ritorna NULL. Tipici errori che si possono avere nell'apertura di un file sono, per esempio:

- il tentativo di aprire in lettura un file che non esiste;
- aprire in scrittura un file read-only (es. i file su un CD-ROM);
- aprire un file per il quale non si dispongono dei necessari permessi di accesso.

Per esempio, le linee di codice seguenti

```
FILE *fin, *fout;

if (!(fin = fopen("matrice.dat", "r"))) {
  perror("matrice.dat");
  exit(1);
}
if (!(fout = fopen("documenti/info.txt", "w"))) {
  perror("documenti/info.txt");
  exit(1);
}
```

aprono rispettivamente il file matrice.dat in lettura, assegnando il valore alla variabile fin e il file info.txt nella sottodirectory documenti in scrittura, assegnando il puntarore fout. Le istruzioni condizionali controllano che i puntatori ritornati siano non nulli, ovvero che ciascun file sia stato aperto correttamente. In caso di errore, viene stampato un messaggio e il programma viene terminato.

12.3.2 Chiusura di file

Una volta che il file è stato acceduto in lettura e/o scrittura, e non è più necessario accedervi, il file deve essere *chiuso* utilizzando la funzione fclose, la quale è dichiarata in stdio.h come segue

```
int fclose(FILE *fp);
```

La funzione accetta come parametro il puntatore a FILE che identifica il file da chiudere. Essa ritorna 0 se la chiusura avviene con successo, oppure il valore EOF in caso di errore.

La funzione serve per liberare (flush) i buffer interni della libreria nella quale sono state memorizzate le informazioni lette e scritte sul file (vedi sezione 12.11.1 sulla bufferizzazione dell'I/O).

12.4 FORZARE LA SCRITTURA DEI DATI CON FFLUSH

Quando i dati vengono scritti su un file utilizzando le funzioni del C, questi non sono effettivamente scritti su disco, ma vengono conservati in memoria per questioni di efficienza nell'accesso al disco fisico. Per forzare la scrittura su disco dei dati, è possibile utilizzare la funzione fflush, dichiarata come segue:

```
int fflush(FILE *stream);
```

che accetta come parametro il puntatore a FILE stream da scrivere.

12.5 ACCESSO A FILE BINARI: LE FUNZIONI FREAD E FWRITE

Le funzioni `fread` e `fwrite` permettono di leggere e scrivere su `FILE` dati in formato binario. Sono definite come segue:

```
size_t fwrite(void* pt, size_t size, size_t nelem, FILE* f);
size_t fread(void* pt, size_t size, size_t nelem, FILE* f);
```

I parametri utilizzati sono:

- `pt` è l'indirizzo di memoria al quale sono memorizzati i dati da scrivere su file o al quale scrivere i dati letti da file;
- `size` è la dimensione del dato singolo;
- `nelem` è il numero di dati da trattare, ovvero il numero di blocchi di dimensione `size` da scrivere/leggere;
- `f` è il puntatore a `FILE` che identifica il file in cui scrivere (leggere).

Le funzioni `fread` e `fwrite` ritornano un numero intero che indica quanti elementi sono stati rispettivamente letti e scritti con successo. Infatti, è possibile che le funzioni leggano o scrivano un numero minore di dati di quelli specificati: la `fread` legge un numero inferiore di dati in genere quando viene raggiunta la fine del file di input (un valore di ritorno minore di `nelem` viene infatti spesso usato per controllare il raggiungimento della fine del file), mentre la `fwrite` potrebbe scrivere un numero minore di valori in caso di esaurimento dello spazio su disco. È quindi consigliabile controllare sempre il valore di ritorno delle funzioni, al fine di prevenire eventuali errori dovuti a scritture/letture parziali e gestire opportunamente le relative condizioni anomale.

Il programma seguente effettua la copia di un file binario di interi.

Una versione estesa del programma è contenuto nel file `frw.c`.

```
1  #include <stdio.h>
2
3  int main(int argc, char ** argv)
4  {
5    int buf[1024];
6    int n;
7    FILE *fin, *fout;
8
9    if (argc != 3) return 1;
10
11   if (!(fin = fopen(argv[1], "r"))) {
12     perror(argv[1]);
13     exit(1);
14   }
15   if (!(fout = fopen(argv[2], "w"))) {
16     perror(argv[2]);
17     exit(2);
18   }
19   do {
20     n = fread(buf, sizeof(int), 1024, fin);
```

```
21      fwrite(buf, sizeof(int), n, fout);
22    } while (n);
23
24    fclose(fin);
25    fclose(fout);
26    return 0;
27  }
```

Il programma dichiara con l'istruzione 7 due variabili di tipo puntatore a `FILE`, `fin` e `fout`, che individueranno il file da copiare ed il file destinazione.

Le due variabili sono inizializzate chiamando la funzione di libreria `fopen` alle linee 11 e 15 che aprono rispettivamente un file in lettura e uno in scrittura. I nomi dei file vengono ottenuti dalla linea di comando, e sono rispettivamente il primo e il secondo parametro[21]. Le relative istruzioni condizionali controllano che il puntatore ritornato sia non nullo, ovvero che il file sia stato aperto correttamente. In caso di errore, viene stampato un messaggio e il programma viene terminato.

Il ciclo `do-while` continua ad eseguire fintanto che il valore di `n` è diverso da 0, ovvero fintanto che l'istruzione `fread` alla linea 20 legge un numero di elementi diverso da 0.

Con tale istruzione, infatti, vengono letti 1024 interi, ciascuno dei quali ha dimensione `sizeof(int)`, dal file `fin`, e memorizzati nel vettore `buf` di dimensione opportuna. Il significato dell'istruzione è quella di leggere *al più* 1024 elementi. Se una lettura dal file non permette di leggerne tanti, semplicemente la funzione ritorna il numero di elementi *effettivamente letti*.

Successivamente, l'istruzione `fwrite` alla linea 21 scrive nel file `fout` il numero `n` di elementi letti, prelevandoli dal buffer `buf`.

Al termine del ciclo, i file vengono chiusi utilizzando la funzione `fclose` (linee 24 e 25). È importante chiudere un file quando esso viene scritto, altrimenti i dati potrebbero non venir memorizzati al termine del programma di scrittura. Per quanto riguarda il file in lettura, è sempre bene chiuderlo esplicitamente, anche se l'omissione dell'istruzione di chiusura non causa in genere problemi particolari.

12.6 LETTURA DI FILE DI TESTO

Per l'interazione con l'utente, quando sia necessario leggere dei dati dalla console, è preferibile usare una tecnica basata sull'uso della funzione `fgets`[22] e `sscanf` invece che utilizzare la funzione `scanf`.

La funzione `fgets` è dichiarata come segue:

```
char *fgets(char *s, int size, FILE *stream);
```

i parametri hanno il seguente significato

- `s` è il puntatore alla stringa letta;
- `size` è il numero massimo di caratteri da restituire;
- `stream` è il file dal quale leggere la stringa (porlo pari a `stdin` per leggere da tastiera).

il valore di ritorno è il puntatore alla stringa stessa.

La funzione `sscanf` è invece dichiarata come segue:

```
int sscanf(const char *str, const char *format, ...);
```

[21] La prima stringa nel vettore `argv` è il nome del programma lanciato.

[22] E' disponibile anche la funzione `gets`, ma il suo utilizzo è sconsigliato in quanto non permette di specificare il numero massimo di caratteri da restituire; può quindi presentare problemi di sicurezza e stabilità dei programmi, a causa dei possibili buffer overflow.

e il suo funzionamento è esattamente identico a quello della `scanf`[23], soltanto che invece di fare il parsing dell'input proveniente direttamente dalla console `stdin`, viene analizzata la stringa `str`, cioè il primo parametro della funzione.

Il programma di esempio seguente mostra l'utilizzo della coppia di funzioni `fgets` e `sscanf`:

Il programma è contenuto nel file `voti.c`.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #define MAX_STUD  (5)
5
6  struct studente {
7    char nome[80];
8    int voto;
9  };
10
11 struct studente stud[MAX_STUD];
12
13 int main()
14 {
15   char s[100];
16   int cont = 0;
17   int err = 0;
18   int somma = 0;
19
20   while ((fgets(s, sizeof(s), stdin)) && (cont < MAX_STUD)) {
21     sscanf(s, "%80s %d", stud[cont].nome, &stud[cont].voto);
22     if (stud[cont].voto >= 18 && stud[cont].voto <= 30) {
23       somma += stud[cont].voto;
24       cont++;
25     } else err++;
26   }
27
28   if (cont > 0)
29     somma /= cont;
30
31   printf("studenti %d, media %d errori %d\n", cont, somma, err);
32
33   return 0;
34 }
```

Il programma legge i nominativi di studenti e i loro voti in varie prove. I dati sono contenutiin stringhe di testo lette con `fgets` dalla console (riga 20), ovvero dal file di input è `stdin`, di lunghezza massima pari a 100 caratteri. Si noti il prezioso uso dell'operatore `sizeof` nella chiamata a `fgets`. La stringa letta viene analizzata con `sscanf` della linea 21, la quale si aspetta il nome dello studente, cioè una stringa, e il voto, cioè un intero. Il ciclo continua fintanto che non viene immesso il carattere `EOF`

[23] Attenzione alla "s" in più!

(End Of File), che da tastiera equivale ad immettere il carattere Ctrl-D[24] su sistemi UNIX o Ctrl-Z su sistemi DOS, oppure finché il conteggio del numero di informazioni lette non supera la soglia massima memorizzabile e impostata mediante `MAX_STUD` (linea 4). Nel ciclo `while` viene tenuta traccia del numero di studenti e della media complessiva dei loro voti. I dati vengono memorizzati nel vettore di strutture `stud` soltanto se il voto immesso è valido, ovvero se è un intero compreso tra 18 e 30. Inoltre viene tenuta traccia del numero di immissioni non valide.

12.7 REDIREZIONE DELL'INPUT E DELL'OUTPUT

Le funzioni di I/O come `fgets`, `fputs`, `fscanf`, `fprintf` e le altre funzioni che leggono e scrivono su file, permettono la possibilità di redirigere l'input e l'output al momento dell'invocazione del programma da linea di comando.

È sempre possibile utilizzare le funzioni indicate in sostituzione delle funzioni `scanf` e `printf` già ampiamente utilizzate, semplicemente indicando che letture e scritture avvengono sui file standard `stdin` e `stdout`. Per esempio, le coppie di istruzioni seguenti sono equivalenti, quando l'interazione utente avviene tramite tastiera e video:

```
scanf("%d %f %s", &intero, &numero, stringa);
fscanf(stdin, "%d %f %s", &intero, &numero, stringa);

printf("%d %f %s\n", intero, numero, stringa);
fprintf(stdout, "%d %f %s\n", intero, numero, stringa);
```

Quando vengono utilizzate tali funzioni per la lettura/scrittura dei dati, è possibilie sfruttare la tecnica della redirezione dell'input/output da linea di comando, per fornire i valori di ingresso ad un programma e per specificare un file di output alternativo ad `stdout`. Questo utile espediente sfrutta l'associazione tra il file standard `stdin` e il terminale (tipicamente la tastiera), e tra il file standard `stdout` e la console (il video). Un esempio di comando di redirezione è il seguente:

```
# ./matrix_strtok < matrix.dat
```

Per ottenere l'output su file, invece che sulla console, il comando è:

```
# ./matrix_strtok < matrix.dat > outfile.dat
```

il quale riversa l'output sul file `outfile.dat`.

È possibile creare dei programmi che leggano il proprio input dal file `stdin` e poi, al momento di invocare il programma, usare l'operatore di redirezione dell'input della shell per redirigere il contenuto di un file sullo standard input del programma. Molti programmi di sistema della shell di UNIX, come `cat`, possono essere usati efficacemente in questo modo.

Allo stesso modo, i programmi che scrivono i dati su `stdout` possono essere rediretti per scrivere i valori di uscita su un file generico.

12.7.1 Il programma `count.c`

Il programma `count.c` conta il numero di caratteri che vengono letti dallo standard input.

Il programma è contenuto nel file `count.c`. Il file `count.txt` contiene una matrice di esempio da utilizzare in combinazione con la redirezione dell'input.

[24]Tenere premuti contemporaneamente i tasti Control e D.

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

int main()
{
  char s[100];
  int count = 0;

  while (fgets(s, sizeof(s), stdin))
    count += strlen(s);

  printf("%d\n",count);

  return 0;
}
```

Si noti come non sia presente alcuna istruzione `fopen` per l'apertura di file, in quanto il file `stdin` è aperto automaticamente all'avvio del programma.

L'istruzione 9 continua a leggere una stringa dal file `stdin` fintanto che non viene inserito il carattere di End Of File (EOF), che da tastiera equivale al carattere di controllo `Control-D` (o ^D). Quando ciò accade, la funzione ritorna un puntatore nullo e il programma termina stampando il numero totale di caratteri letti, memorizzato ad ogni ciclo incrementando la variabile `count` alla linea 10. La funzione `fgets` legge una stringa di *al più* `sizeof(s)` caratteri, memorizzandoli nella variabile `s`.

Una volta compilato, il programma può essere invocato da linea di comando come segue:

```
$ ./count
Queste<RETURN>
sono stringhe<RETURN>
lette<RETURN>
dallo standard input.<^D>
49
$
```

dove i <`RETURN`> e il ^`D` corrispondono ai relativi tasti. Dopo l'invocazione del programma `count`, le prime 4 stringhe sono inserite da tastiera, mentre il numero 49 viene stampato dal programma una volta che l'input è stato terminato dal carattere ^`D`.

Ora, se il file `count.txt` contiene esattamente le stesse stringhe:

```
Queste
sono stringhe
lette
dallo standard input.
```

allora l'istruzione

```
$ ./count < count.txt
49
$
```

produce esattamente lo stesso risultato, poiché il contenuto del file `count.txt` viene rediretto sullo standard input del programma.

Similmente è possibile redirigere l'output su file con un'istruzione del tipo

```
$ ./count < count.txt > output.dat
$ cat output.dat
49
$
```

la quale scrive sul file `output.dat` tutto ciò che normalmente verrebbe scritto sul video. Si noti che l'esecuzione del programma `count` non genera nessun messaggio su video, ma tutto l'output viene scritto su `output.dat`, successivamente visualizzato con il comando `cat`.

Ovviamente è possibile redirigere anche il solo output, con un comando del tipo:

```
$ ./count > output.dat
```

In tal caso l'inserimento dei dati avviene da tastiera, mentre i messaggi di output vengono scritti sul file `output.dat`.

12.7.2 Un altro esempio di redirezione da linea di comando

Riprendendo l'esempio riportato in Sezione 12.6, supponiamo di generare un esegubile chiamato `voti`. Il risultato di una esecuzione del tipo

```
$ ./voti
```

fa in modo che il programma attenda l'inserimento dei dati da tastiera linea per linea. Ciascuna linea è separata da una andata a capo (tasto `INVIO`), mentre la sequenza di linee di ingresso viene terminata con l'immissione del carattere `Ctrl-D` (o `Ctrl-Z` su sistemi DOS).

La redirezione dell'I/O prevede di avere un file nel quale sono memorizzati i dati. Supponendo che il file `voti.dat` sia il seguente:

Il file di dati è `voti.dat`.

```
$ cat voti.dat
Bianchi 30
Rossi 25
Ferrari 27
Marconi 26
Facchini 16
```

è possibile richiamare il programma sfruttando la redirezione dell'input permessa dalla shell con il seguente comando

```
$ ./voti < voti.dat
studenti 4, media 27 errori 1
$
```

ottenendo il relativo output. Il risultato sarebbe identico anche introducendo gli stessi dati da tastiera. Ovviamente, dal momento che il semplice programma di esempio non utilizza il nome degli studenti in alcun modo, il risultato potrebbe essere identico anche variando il nome degli studenti.

Inoltre, è possibile redirigere l'output nel modo seguente:

```
$ ./voti < voti.dat > risultati.dat
$ cat risultati.dat
studenti 4, media 27 errori 1
$
```

la quale scrive i messaggi di output sul file `risultati.dat` invece che sulla console.

12.8 LETTURA DI FILE STRUTTURATI CON FGETS E SSCANF

Quando un file ha una struttura nota, e tipicamente di tratta di file di testo che contengono informazioni strutturate, è possibile utilizzare lo schema di lettura basato su `fgets` e `sscanf`.

Il programma seguente legge da standard input, con possibilità di redirezione dello standard input, una matrice formata da 5 righe, ciascuna delle quali contiene 4 numeri in virgola mobile.

Il programma è contenuto nel file `matrix_fgets_sscanf.c`. Il file `matrix.dat` contiene una matrice di esempio da utilizzare in combinazione con la redirezione dell'input.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #define N (5)
5
6  int main()
7  {
8    char s[100];
9    float m[N][4];
10   int i;
11
12   i = 0;
13   while (fgets(s, sizeof(s), stdin)) {
14     sscanf(s, "%f %f %f %f", &m[i][0], &m[i][1], &m[i][2], &m[i][3]);
15     printf("%f %f %f %f\n", m[i][0], m[i][1], m[i][2], m[i][3]);
16     i++;
17   }
18
19   return 0;
20 }
```

Da notare che il programma non effettua controlli sul fatto che vengano lette più righe di quelle memorizzabili nella matrice. Nel caso che il numero di record inseriti superi il massimo memorizzabile, si ha un overflow con tutti i relativi possibili inconvenienti. La (banale) modifica per gestire tale tipo di errore è lasciata al lettore.

Un esempio di utilizzo del programma in combinazione con la redirezione dell'input è la seguente:

```
# ./matrix_fgets_sscanf < matrix.dat
```

Questo metodo è molto comodo per effettuare il debug di programmi che attendono l'input da tastiera o da file.

12.9 I/O CON LE FUNZIONI FSCANF E FPRINTF

Quando un file da leggere o scrivere ha una struttura nota è possibile anche usare le istruzioni `fscanf` e `fprintf` per la lettura e per la scrittura.

Queste funzioni si comportano come le funzioni `scanf` e `printf`, già più volte utilizzate per la lettura da tastiera e l'output a video. La differenza consiste nel fatto che è possibile specificare su quale file leggere e scrivere. Si possono utilizzare anche i file standard `stdin` e `stdout`; in questo caso il comportamento è identico a quello di `scanf` e `printf`.

Le due funzioni sono definite come segue:

```
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

La differenza con le funzioni `scanf` e `printf` è che il primo parametro rappresenta il file sul quale scrivere (o leggere).

Il programma seguente legge da standard input, con possibilità di redirezione dello standard input, una matrice di 4 colonne per un massimo di 5 righe. Il programma scrive su standard output la matrice moltiplicata per 2.

Il programma è contenuto nel file `matrix_fscanf_fprintf.c`. Il file `matrix.dat` contiene una matrice di esempio da utilizzare in combinazione con la redirezione dell'input.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #define N (5)
5
6  int main()
7  {
8    float m[N][4];
9    int i;
10
11   i = 0;
12   while (!feof(stdin)) {
13     fscanf(stdin, "%f %f %f %f",
14       &m[i][0], &m[i][1], &m[i][2], &m[i][3]);
15     fprintf(stdout, "%f %f %f %f\n",
16       2 * m[i][0], 2 * m[i][1], 2 * m[i][2], 2 * m[i][3]);
17     i++;
18   }
19
20   return 0;
21 }
```

Per controllare la terminazione dell'input viene utilizzata la funzione `feof`, la quale ritorna non-zero se l'indicatore di fine file è impostato.

Anche in questo caso, il programma non effettua controlli sul fatto che il numero di righe lette sia minore o uguale della dimensione della matrice.

12.10 ESEMPIO DI LETTURA DI MATRICI CON `FGETS` E `SSCANF`

I programmi illustrati nelle sezioni 12.8 e 12.9 assumevano che il numero di colonne fosse noto a priori, e quindi era possibile impostare l'istruzione `sscanf` of `fscanf` con il numero appropriato di parametri per leggere tutti i dati su ciascuna linea.

Quando il numero di parametri su una linea non è noto a priori oppure è sconveniente elencare tutti i parametri uno per uno, è possibile utilizzare un approccio più adatto e generale.

Il tipico esempio è quello della lettura di una matrice da un file (o standard input) che non abbia un numero di colonne noto a priori. In tal caso, non è possibile elencare esplicitamente le variabili nell'istruzione di scansione della stringa di input.

Il programma seguente legge da standard input, con possibilità di redirezione dello standard input, una matrice avente un numero arbitrario di elementi per ciascuna riga. Il numero di elementi è

comunque limitato a 10, a causa della dichiarazione statica della matrice che deve contenere i dati. Anche in questo caso, non sono effettuati i controlli necessari a verificare che si leggano più righe e colonne di quanto sia possibile memorizzare nella matrice, ma le modifiche da apportare sono semplici e vengono lasciate come esercizio per il lettore.

Il programma è contenuto nel file `matrix_strtok.c`. Il file `matrix.dat` contiene una matrice di esempio da utilizzare in combinazione con la redirezione dell'input.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #define MAX_N (10)
5
6  int main()
7  {
8    char s[100], *s1;
9    float m[MAX_N][MAX_N];
10   int i, j;
11
12   i = 0;
13   while (fgets(s, sizeof(s), stdin)) {
14     j = 0;
15     if ((s1 = strtok(s, " ")) != NULL) {
16       sscanf(s1, "%f", &m[i][j]);
17       printf("%f ", m[i][j]);
18       j++;
19     } else {
20       printf("errore nel parsing dei valori\n");
21       return 1;
22     }
23     while ((s1 = strtok(NULL, " ")) != NULL) {
24       sscanf(s1, "%f", &m[i][j]);
25       printf("%f ", m[i][j]);
26       j++;
27     }
28     printf("\n");
29     i++;
30   }
31
32   return 0;
33 }
```

Il programma legge le stringhe dal file di ingresso alla linea 13 con l'istruzione `fgets`, corrispondenti ad una riga della matrice. Per isolare gli elementi di ciascuna colonna, viene usata la funzione di libreria `strtok` (linee 15 e 23), la quale suddivide una data stringa in un insieme di token. Un token si può definire come una sottostringa della stringa da partizionare, che sia delimitato da un opportuno delimitatore. Per esempio, la stringa

```
"10 40 3.5 78 1"
```

viene suddivisa nei token

```
"10" "40" "3.5" "78" "1"
```

Nell'esempio, i singoli valori numerici sono delimitati da spazi: i delimitatori sono quindi gli spazi bianchi, mentre i singoli token sono le sottostringhe che rappresentano i valori numerici.

La prima volta che la funzione `strtok` viene chiamata (linea 15), riceve in ingresso la stringa da decomporre in token (`s` nell'esempio), e il separatore dei token, che nell'esempio è uno spazio. Ogni successiva chiamata a `strtok` deve passare `NULL` come stringa, poiché questa è stata specificata nella prima chiamata. Ecco perché nella chiamata ad `strtok` alla linea 23 il primo parametro è `NULL`. Il ciclo continua fintanto che `strtok` ritorna un puntatore valido (non nullo) che rappresenta il puntatore al primo carattere del token successivo. Gli indici che servono per tenere traccia di quale elemento viene assegnato nella matrice sono opportunamente incrementati per far avanzare il numero di riga (linea 29) e colonna (linea 26).

12.11 FSCANF E FGETS A CONFRONTO

La lettura di dati strutturati da un file è possibile sia utilizzando la funzione `fscanf` che l'accoppiata di funzioni `fgets` e `sscanf`.

Infatti, se è necessario leggere dal file `fin` delle stringhe di caratteri nelle quali sono presenti 3 numeri in virgola mobile, i due metodi alternativi seguenti sono equivalenti:

```
fscanf(fin, "%f %f %f", &n1, &n2, &n3);
```

oppure

```
fgets(buf, sizeof(buf), fin);
sscanf(buf, "%f %f %f", &n1, &n2, &n3);
```

dove `buf` è un opportuno buffer (es. `char buf[100]`) per la memorizzazione della stringa in ingresso.

Ci sono però valide ragioni per cui è sempre bene utilizzare le funzioni `fgets` e `sscanf` invece della singola `fscanf`. In particolare, le motivazioni proposte riguardano da un lato la bufferizzazione dell'input e dall'altro sono ragioni legate alla sicurezza del programma generato, dal punto di vista di possibili attacchi informatici al sistema che utilizza il programma. Queste motivazioni vengono illustrate rispettivamente nelle sezioni 12.11.1 e 12.11.2.

12.11.1 La bufferizzazione dell'input

Il problema della bufferizzazione dell'input si presenta quando vengono utilizzate funzioni come `scanf` e `fscanf`. Il problema è dovuto al fatto che per tali funzioni i caratteri "spazio" e "andata a capo" (INVIO) sono sostanzialmente intercambiabili.

Quando si inseriscono infatti gli elementi che vengono attesi dalla `scanf`, è possibile separare i singoli valori sia mediante lo spazio che l'invio. Questo crea un evidente problema, per due motivi:

1. l'utente che inserisce i dati si aspetta in genere che il proprio input termini quando si preme invio;

2. se vengono inseriti più dati di quelli che sono attesi, questi vengono mantenuti in un buffer e letti, insieme agli altri eventualmente inseriti, nella successiva operazione di lettura.

In particolare, il mantenimento dei dati in un buffer di ingresso, fa si che inserendo più elementi di quelli attesi da una istruzione – cosa possibile separando gli elementi con degli spazi – gli elementi in eccesso sono letti dalle successive istruzioni, creando confusione e potenziali pericolosi errori nell'assegnamento delle variabili.

Si consideri il seguente programma di esempio

Il programma è contenuto nel file `buf_scanf.c`.

```c
#include <stdio.h>

int main()
{
  int n1, n2, n3, n4, n5, n6;

  scanf("%d %d %d", &n1, &n2, &n3);
  printf("valori inseriti: %d %d %d\n", n1, n2, n3);

  scanf("%d %d %d", &n4, &n5, &n6);
  printf("valori inseriti: %d %d %d\n", n4, n5, n6);

  return 0;
}
```

Esso si attende l'inserimento di 3 valori interi, che vengono poi stampati, e successivamente l'inserimento di altri tre valori interi, a loro volta poi stampati. Si considerino dunque le seguenti esecuzioni del programma, nelle quali i numeri sono separati con varie combinazioni di spazi e INVIO.

```
$ ./buf_scanf
1 2 3
valori inseriti: 1 2 3
4 5 6
valori inseriti: 4 5 6
```

Nell'esempio i primi tre valori sono separati da spazi e terminati dall'INVIO come ci si aspetterebbe. Allo stesso modo, nell'esempio seguente si separa ciascun valore con INVIO e si ottiene lo stesso output:

```
$ ./buf_scanf
1
2
3
valori inseriti: 1 2 3
4
5
6
valori inseriti: 4 5 6
```

Nell'esempio seguente, invece, si forniscono più valori per la prima funzione:

```
$ ./buf_scanf
1 2 3 4
valori inseriti: 1 2 3
4 5 6
valori inseriti: 4 4 5
```

si noti che l'ultimo valore inserito nella prima istruzione, viene assegnato alla prima variabile nella seconda istruzione, che potrebbe non essere di facile interpretazione. Ci si potrebbe infatti aspettare che il valore venisse scartato, mentre invece produce un assegnamento diverso delle variabili rispetto agli esempi precedenti.

Lo stesso tipo di input può essere ottenuto utilizzando opportunamente la `fgets`, come nell'esempio seguente

Il programma è contenuto nel file buf_fgets.c.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    char s[100];
6    int n1, n2, n3, n4, n5, n6;
7
8    fgets(s, sizeof(s), stdin);
9    sscanf(s, "%d %d %d", &n1, &n2, &n3);
10   printf("valori inseriti: %d %d %d\n", n1, n2, n3);
11
12   fgets(s, sizeof(s), stdin);
13   sscanf(s, "%d %d %d", &n4, &n5, &n6);
14   printf("valori inseriti: %d %d %d\n", n4, n5, n6);
15
16   return 0;
17 }
```

Il primo vantaggio della fgets è quello che una stringa viene sempre terminata solo dall'INVIO. Questo comporta dei sicuri vantaggi in termini di chiarezza dell'input quando ci sono delle istruzioni di letture in sequenza. D'altra parte, se si inseriscono meno dati di quelli attesi, quelli non inseriti assumono valori casuali, come nell'esempio che segue:

```
$ ./buf_fgets
1
valori inseriti: 1 32768 0
2 3
valori inseriti: 2 3 1970169159
```

Il problema è però gestibile inizializzando opportunamente i dati prima della lettura.

Se invece si inseriscono più valori di quelli attesi, come nell'esempio seguente:

```
$ ./buf_fgets
1 2 3 4
valori inseriti: 1 2 3
4 5 6
valori inseriti: 4 5 6
```

semplicemente questi vengono scartati, senza possibilità di "inquinare" successive operazioni di lettura. Il valore in eccesso nella prima operazione di lettura è stato scartato.

12.11.2 Il problema del buffer overflow

Il problema di sicurezza che si può presentare utilizzando la funzione scanf o fscanf è il cosiddetto "buffer overflow", ovvero la scrittura in un buffer che è sottodimensionato rispetto ai dati che vi vengono scritti. I dati in eccesso sono scritti comunque in memoria, e vanno a sovrascrivere l'area di memoria che segue la variabile che costituisce il buffer. Questo può potenzialmente permettere di scrivere del codice macchina in memoria, che poi potrebbe essere eseguito all'insaputa dell'utilizzatore della macchina.

Per esempio, il codice:

```
if (fscanf (fin, "%s", buff) == 1) {
   /* ... */
}
```

può creare problemi, dal momento che la `fscanf` legge dati fintanto che non trova un EOF o un carattere di andata a capo. Quindi, *indipendentemente dalla dimensione di `buff`*, verranno copiati in `buff` stesso tanti caratteri quanti sono quelli letti, causando per l'appunto un *overflow* del buffer stesso. Il problema nasce dal fatto che la `fscanf` non effettua nessun controllo sul numero di caratteri letti.

Per ovviare a tale problema, il codice precedente può essere sostituito con

```
1  {
2    char buff[BUFSIZ], st[BUFSIZ];
3
4    if ((fgets(buff, sizeof(buff), fin) == NULL) ||
5        sscanf (buff, "%s", st) != 1)
6    {
7      /* ... */
8    }
9  }
```

A differenza di `fscanf`, la `fgets` riceve come argomento il *numero massimo di caratteri da leggere*, e quindi non rischia - se utilizzata correttamente come nell'esempio - di provocare overflow. Infatti, nel buffer `buff` saranno memorizzati al più `sizeof(buff)`, senza rischio di causare overflow.

Capitolo 13

PROGRAMMI COMPOSTI DA PIÙ FILE SORGENTE

Programmi di grosse dimensioni possono richiedere molte funzionalità diverse. A loro volta, ciascuna funzionalità può essere implementata per mezzo di collezioni anche molto numerose di funzioni. Quando la complessità del programma, sostanzialmente misurabile in linee di codice o in numero di funzioni, cresce oltre un certo limite, l'utilizzo di un unico file sorgente per l'intero programma diventa problematico per due ragioni principali:

- ogni modifica anche minimale al programma richiede la compilazione di tutto il sorgente
- diventa difficile cercare una determinata porzione di codice all'interno di un sorgente molto lungo e complesso

In una situazione del genere, diventa importante suddividere opportunamente il sorgente in più file diversi. Generalmente una buona linea guida per tale suddivisione è quella di raggruppare le funzioni e i dati relativi ad una determinata funzionalità in un singolo file sorgente. D'altra parte, nulla vieta di implementare ogni singola funzione in un file sorgente dedicato. La suddivisione del programma in file separati risolve i problemi illustrati precedentemente: la ricerca della porzione di codice relativa ad una certa funzionalità richiede di analizzare solo il file sorgente corrispondente. Ancora più importante è il guadagno in termini di velocità di compilazione, dal momento che una piccola modifica al programma richiede la rigenerazione del file oggetto corrispondente al solo file sorgente modificato.

13.1 ESEMPIO DI UTILIZZO DI PIÙ FILE SORGENTE

Si supponga per esempio che il programma eseguibile

```
prog
```

sia stato scritto utilizzando due funzioni `func1` e `func2` poste rispettivamente all'interno dei due file sorgente:

- `func1.c`
- `func2.c`

mentre il file `prog.c` contiene la funzione `main` che richiama le due funzioni. Il contenuto dei diversi file è riportato di seguito. Il file `prog.c` è il seguente:

I file che compongono il programma illustrato nell'esempio sono contenuti nei file `prog.c`, `prog.h`, `func1.c` e `func2.c`.

```
1  #include <stdio.h>
2  #include "prog.h"
3
4  int main() {
5
6    printf("%s: %s %d\n", __FILE__, __FUNCTION__, __LINE__);
7    func1();
8
9    printf("%s: %s %d\n", __FILE__, __FUNCTION__, __LINE__);
10   func2();
11
12   return 0;
13 }
```

Il file `func1.c` è il seguente:

```
1  #include <stdio.h>
2  #include "prog.h"
3
4  int func1() {
5    printf("%s: %s %d\n", __FILE__, __FUNCTION__, __LINE__);
6
7    return 0;
8  }
```

Il file `func2.c` è il seguente:

```
1  #include <stdio.h>
2  #include "prog.h"
3
4  int func2() {
5    printf("%s: %s %d\n", __FILE__, __FUNCTION__, __LINE__);
6
7    return 0;
8  }
```

Le varie funzioni contengono una sola istruzione che stampa le tre variabili

```
__FILE__
__FUNCTION__
__LINE__
```

tali variabili sono assegnate in fase di compilazione e rappresentano rispettivamente

- il nome del file nel quale l'istruzione che usa la variabile è posta;
- il nome della funzione nella quale l'istruzione che usa la variabile è posta;
- il numero di linea alla quale l'istruzione che usa la variabile è posta;

I programmi, oltre al file di intestazione standard `stdio.h`, includono anche il file di intestazione realizzato per questo specifico programma, ovvero `prog.h`. Si noti che quest'ultimo viene incluso racchiudendo il nome del file tra doppi apici, per indicare al preprocessore di ricercare tale file nella directory corrente invece che nelle directory di sistema. Il file `prog.h` contiene le seguenti istruzioni:

```
1 #ifndef __PROG_H__
2 #define __PROG_H__
3 int func1();
4 int func2();
5 #endif
```

cioè contiene le dichiarazioni delle funzioni definite nei diversi file `.c`. Questa prassi, che prevede di elencare in un file di header tutte le funzioni e le eventuali variabili `extern` usate in un programma, permette di poter usare le funzioni scrivendone la dichiarazione una volta sola, grazie al fatto che il file di intestazione viene poi incluso in tutti i file che fanno uso di tutte o di parte delle funzioni.

Altra prassi molto comune nella scrittura di file di intestazione ad hoc è quella di racchiudere tutto il codice all'interno delle direttive del preprocessore del tipo

```
#ifndef __PROG_H__
#define __PROG_H__
  /* dichiarazioni */
#endif
```

dove la macro che viene definita si riferisce al nome del file header. In questo caso si chiama `__PROG_H__` in quanto il file di intestazione è `prog.h`. Il codice verrà quindi incluso soltanto se `__PROG_H__` NON è già stata definita. La macro viene infatti definita la prima volta che il codice viene incluso, a causa della presenza della

```
#define __PROG_H__
```

che compare subito dopo la `#ifndef`.

Questo evita che si abbia duplicazione di codice che in alcuni casi può portare ad errori di compilazione. La duplicazione può avvenire in programmi complessi quando i file di intestazione vengono inclusi in altri file di intestazione. Questo è il modo migliore per non essere costretti a tenere traccia di quali header vengono inclusi in determinati file ed evitare problemi di compilazione.

L'output del programma è quindi il seguente:

```
$ ./prog
prog.c: main 5
func1.c: func1 4
prog.c: main 8
func2.c: func2 4
```

Il punto importante da comprendere è come sia possibile compilare il programma finale `prog` a partire dai singoli file sorgente. Le istruzioni da impartire per la compilazione sono le seguenti:

```
$ gcc -c -o func1.o func1.c
$ gcc -c -o func2.o func2.c
```

```
$ gcc -o prog prog.c func1.o func2.o
```

La prima (seconda) istruzione richiede al compilatore di compilare il file sorgente `func1.c` (`func2.c`) e di produrre il file `func1.o` (`func2.o`). Il parametro `-c` fornito alla linea di comando indica al compilatore di effettuare la sola compilazione ma non il linking. Senza questa opzione il compilatore segnalerebbe un errore, in quanto non è in grado di trovare la funzione `main`. Infatti, i due file non sono programmi completi, in quanto non contengono la funzione `main`, per cui si ha la segnalazione di errore da parte del compilatore.

La terza istruzione richiede al compilatore di compilare il file sorgente `prog.c` e di effettuare il linking linkando il contenuto dei file oggetto `func1.o` e `func2.o`. Se infatti si tentasse di compilate il programma `prog.c` senza linkare i due file, il compilatore segnalerebbe un errore in quanto non sarebbe in grado di trovare il codice relativo alle due funzioni. Può infatti essere interessante verificare cosa accade se si tenta di compilare il programma `prog.c` senza linkare opportunamente i file contenenti le funzioni utilizzate nel `main`:

```
# gcc -Wall prog.c -o prog
/tmp/ccGzBrKM.o: In function 'main':
prog.c:(.text+0x28): undefined reference to 'func1'
prog.c:(.text+0x50): undefined reference to 'func2'
collect2: ld returned 1 exit status
```

Si nota che il linker genera un messaggio di errore di `undefined reference`, poiché nel `main` si fa riferimento a due funzioni delle quali non si trova il codice oggetto. Il file

```
/tmp/ccGzBrKM.o
```

è il file temporaneo, con un nome generato casualmente dal compilatore, che viene utilizzato per la generazione del codice oggetto del file contenente il `main`.

13.2 IL COMANDO `MAKE`

Il comando `make` è una utility che permette di effettuare il building automatico di programmi di grosse dimensioni, ovvero composti da molti file sorgente. In particolare, `make` permette di velocizzare notevolmente il processo di ri-compilazione dell'applicazione qualora siano state effettuate modifiche soltanto ad una parte dei file sorgente.

Comunemente, infatti, in progetti di grosse dimensioni, soltanto pochi file vengono modificati tra una compilazione e la successiva. In questo caso, `make` determina quali sono i file sorgente che devono essere ri-compilati e permette di risparmiare il tempo richiesto da compilazioni non indispensabili.

L'utilizzo di `make` si basa sulla scrittura di una serie di regole che specificano quali sono i file sorgente che compongono l'applicazione, quali sono i comandi che li devono elaborare, e soprattutto quali sono le loro *dipendenze*.

Una dipendenza specifica a `make` quali sono i file che devono essere elaborati per ri-generare il file obiettivo. Il caso tipico è quello dei file oggetto, che dipendono dai rispettivi file sorgente.

La gestione delle dipendenze permette a `make` di determinare automaticamente quali sono i file che devono essere ricompilati sulla base dei tempi di creazione e modifica dei file, impostati dal sistema operativo ogniqualvolta si opera su un file.

`make` rigenera il file target solo se il tempo di modifica di almeno uno dei file dal quale il target dipende *è più recente* del tempo di creazione del file target.

Tipicamente, `make` viene utilizzato per effettuare il building di programmi scritti in C e C++, ma il suo principio di funzionamento è completamente generale, e può quindi essere utilizzato per gestire un qualunque progetto software.

In linea di principio, l'utilizzo di `make` per la generazione del file eseguibile non è strettamente necessaria. D'altra parte, il suo utilizzo si rivela sempre più indispensabile al crescere della complessità del programma, in particolare quando quest'ultimo è composto da molti file sorgente. In tal caso permette di risparmiare molto tempo compilando soltanto i file sorgenti modificati dopo l'ultima compilazione.

13.3 ESEMPIO DI DIPENDENZE

Si supponga per esempio che il programma eseguibile

```
prog
```

sia realizzato come in Sezione 13.1, ovvero sia composto dai tre file `prog.c`, `prog.h`, `func1.c` e `func2.c`.

Il building di `prog` richiede la generazione dei due file oggetto `func1.o` e `func2.o`, ovvero `prog` dipende da `func1.o` e `func2.o`. Inoltre, `prog` dipende anche dal file di intestazione `prog.h`, in quanto se viene modificato quest'ultimo anche il file `prog.c` deve essere ricompilato.

Ciascun file oggetto dipende a sua volta dal rispettivo file sorgente e dal file di intestazione.

Come si vede, le dipendenze creano una gerarchia: il programma dipende dai file oggetto e questi ultimi dipendono dai rispettivi sorgenti. Ciò significa che quando un file sorgente viene modificato, deve essere ri-generato il corrispondente file oggetto per poter generare il nuovo programma.

Il problema risolto automaticamente da `make` è quello di determinare, ad ogni ricompilazione, quali sono i file che devono essere ri-generati e quali no.

Nell'esempio riportato, se dopo aver effettuato con successo un primo building dell'applicazione `prog` viene modificato il file `func2.c`, il tempo di modifica associato a `func2.c` sarà più recente sia di quello di `func2.o` che di `prog`. Quindi un nuovo building ri-genererà soltanto `func2.o` e `prog`, mentre `func1.o` può essere recuperato dalla precedente compilazione.

13.4 IL MAKEFILE

Il *makefile* è un file di testo che viene cercato dal comando `make` quando esso viene invocato, e che contiene le specifiche di dipendenze e comandi per la generazione del programma finale.

Quando `make` viene lanciato, esso cerca il makefile nella directory corrente. Vengono cercati nell'ordine i file `GNUmakefile`, `makefile`, e `Makefile`. Il primo di essi che viene trovato, viene utilizzato come makefile.

È anche possibile specificare un makefile avente un nome qualsiasi, utilizzando l'opzione `-f`. Per esempio:

```
$ make -f makefile.example
```

Le dipendenze tra i file vengono esplicitate nella seguente forma:

```
target: dip_1 ... dip_n
  comando_1
  comando_m
```

Con questa sintassi si indica che il file `target` dipende dai file `dip_1` ... `dip_n`, e viene specificato che per produrre `target` devono essere eseguiti i comandi `comando_1` ... `comando_m`.

Un semplice makefile che permette di costruire il programma di esempio illustrato è il seguente

```
prog: func1.o func2.o prog.c prog.h
```

```
  gcc -o prog prog.c func1.o func2.o

func1.o: func1.c prog.h
  gcc -c -o func1.o func1.c

func2.o: func2.c prog.h
  gcc -c -o func2.o func2.c
```

Nel makefile precedente, si indica al comando `make` che il programma `prog` dipende dai due file oggetto `func1.o` e `func2.o`, oltre che dal file principale `prog.c` e da `prog.h`. A loro volta, i singoli file oggetto dipendono dai relativi file sorgente. Per ciascuna dipendenza viene specificato il comando (potrebbe trattarsi di più comandi) per generare il target a partire dai file dal quale esso dipende.

Nell'esempio considerato, si supponga che il file `prog` sia stato compilato da zero. Se, successivamente, viene modificato uno solo dei due file sorgente contenenti le funzioni, poniamo `func1.c`, quando si lancia il comando

```
$ make prog
```

e si richiede a `make` di rigenerare il target `prog`, verrà prima ricompilato il file `func1.c`, in quanto `func1.o` avrà una data di modifica antecedente a quella della rispettiva dipendenza `func1.c` (`func1.c` è stato modificato dopo l'ultima compilazione).

NOTA

L'indicazione di un comando nella regola *deve essere necessariamente preceduto* da un *carattere di tabulazione*, altrimenti `make` non è in grado di interpretare correttamente la linea di comando, ritenendo che non si tratta, appunto, di un comando.

Capitolo 14

LE LIBRERIE

Le librerie contengono codice e dati, cioè funzioni globali e variabili globali, che possono essere riutilizzati in diversi programmi. Molte funzioni usate dai programmi C sono state standardizzate, così come i nomi degli header da includere prima di usarle. Abbiamo quindi <stdio.h> per lavorare con i file, <string.h> per poter chiamare le funzioni relative alle stringhe (lunghezza, confronto, sottostringa, ...) e moltissimi altri header.

Lo scopo del corso non è quello di prendere confidenza con la molteplicità di funzioni della libreria standard o di altre librerie. Le librerie a disposizione del compilatore sono tantissime e coprono molte delle funzionalità più comunemente richieste. In questo documento sono state già utilizzate delle funzioni di libreria come `printf` o `malloc`. Nonostante l'analisi delle librerie standard non sia l'obiettivo della trattazione, alcune librerie contengono funzioni che verranno descritte nel dettaglio anche in questa sede, in quanto implementano funzionalità che sono ritenute importanti in ogni contesto applicativo. In generale, tutte le funzioni e le variabili globali maggiormente utilizzate (come `stdin` e `stdout`) sono contenute nella *libreria C*, la cosiddetta `libc`, che viene usata automaticamente dal compilatore per risolvere i simboli non definiti nei file sorgente. Il compilatore può disporre anche di una sua propria libreria (per esempio `libgcc`), contenente procedure chiamate dal codice oggetto generato dal compilatore stesso. Anche questa libreria viene inclusa automaticamente durante la fase finale di compilazione.

14.1 USO DI LIBRERIE ESTERNE

Dal momento che molte funzioni sono incluse nelle librerie standard, il programmatore non è tenuto a fornire alcuna informazione aggiuntiva al compilatore e al linker riguardo l'utilizzo di tali librerie, in quanto il compilatore le include automaticamente. Se però è necessario utilizzare una funzione

che è inclusa in una libreria non standard, bisogna esplicitamente istruire il compilatore riguardo alla libreria che contiene la funzione.

Le librerie sono tipicamente memorizzate in file aventi estensione ".a", dette librerie statiche[25]. Le librerie statiche sono create a partire dai file oggetto ".o" per mezzo del programma `ar`, e sono usate dal linker per risolvere i riferimenti alle funzioni al momento della compilazione.

Le librerie standard del C sono solitamente poste nella directory `/usr/lib` e `/lib`. Per esempio, la libreria matematica del C è tipicamente memorizzata nel file `/usr/lib/libm.a`. Le dichiarazioni dei prototipi delle funzioni si trovano nei relativi file di intestazione, che si trovano di solito nella directory `/usr/include`. Per esempio, il file di intestazione della libreria matematica è il file `/usr/include/math.h`. La libreria standard del C è contenuta nel file `/usr/lib/libc.a`, e contiene tutte le funzioni dello standard ANSI/ISO del linguaggio, come `printf`. Questa libreria viene linkata ad ogni programma per default.

In genere, se si tenta di utilizzate una funzione di libreria senza aver esplicitamente istruito il compilatore sul file che contiene le funzioni, il linker genera un errore del tipo

```
# gcc -Wall usemath.c -o usemath
/tmp/ccbR4Ojn.o: In function 'main':
/tmp/ccbR4Ojn.o(.text+0x19): undefined reference
  to 'sqrt'
```

Nel caso citato, il compilatore non è sato in grado di trovare la funzione `sqrt`, utilizzata dal programma `usemath`, dal momento che essa non è definita in `libc.a`[26].

Incidentalmente, il file `/tmp/ccbR4Ojn.o` è un file creato temporaneamente dal compilatore per eseguire l'operazione di linking. Per abilitare il compilatore a linkare la libreria corretta è possibile utilizzare una istruzione come la seguente:

```
# gcc -Wall usemath.c /usr/lib/libm.a -o usemath
```

La libreria `libm.a` contiene i file oggetto delle funzioni matematiche come `sin`, `cos` e `exp` (vedi Sezione 14.5). Il linker cerca tra i file oggetto contenuti nella libreria alla ricerca della funzione da linkare. Quando la funzione desiderata viene trovata, il programma principale può quindi essere correttamente compilato.

Il file eseguibile finale include il codice macchina delle funzioni scritte dal programmatore e di tutte le funzioni di libreria richiamate dal programma.

Per evitare di specificare tutto il percorso di ciascun file di libreria sulla linea di comando, il compilatore prevede una scorciatoia con l'opzione `-l` per linkare le librerie. Per esempio, l'istruzione seguente

```
# gcc -Wall calc.c -lm -o calc
```

è equivalente al comando precedentemente illustrato.

In generale, l'opzione `-lNOME` tenta di linkare i file oggetto contenuti nella libreria di nome `libNOME.a` che si trova nelle directory standard del compilatore. Le directory nelle quali ricercare le librerie possono essere specificate nella linea di comando

Tipicamente la compilazione di un programma includerà varie opzioni di linking del tipo `-lNOME`, per includere tutti i file oggetto necessari, come librerie grafiche, per la comunicazione, ecc.

Per esempio, per utilizzare le funzioni di lettura/scrittura di immagini compresse in formato JPEG, è possibile usare le funzioni della libreria `libjpeg.a` includendo il file di intestazione `<jpeglib.h>` e compilando con

```
# gcc -Wall -ljpeg -o viewer viewer.c
```

[25] Esistono anche le cosiddette *librerie dinamiche*, ma la loro trattazione va oltre gli scopi di queste dispense.
[26] Talvolta la libreria matematica, come altre, vengono linkate per default dal linker.

14.1.1 Il comando ar

Utilizzando il comando ar è possibile creare delle librerie personalizzate, riunendo vari file oggetto in un unico o più file di libreria.

È anche possibile verificare il contenuto di una libreria, sempre utilizzando il comando ar. Digitando per esempio il comando:

```
# ar t /usr/lib/libm.a
```

si ottiene l' output (di cui sotto è mostrata una parte), composto da una serie di nomi di file:

```
...
e_acos.o
e_acosh.o
e_asin.o
e_atan2.o
e_atanh.o
e_cosh.o
e_exp.o
e_fmod.o
...
```

i quali corrispondono ai nomi dei file che contengono il codice compilato delle rispettive funzioni.

14.1.2 Esempio di utilizzo del comando ar

Per illustrare l'utilizzo di librerie esterne si ricorre all'esempio del programma prog presentato nel Capitolo 13, cioè di un programma composto da più file sorgente, che vengono compilati in più file oggetto e quindi linkati per costituire il programma eseguibile.

Il programma prog, il cui file sorgente principale è il file prog.c, richiede l'uso di due funzioni definite nei file func1. e func2.c. Una volta compilati i singoli file oggetto func1.o e func2.o, la generazione dell'eseguibile prevede di linkare tutti i file oggetto con il seguente comando:

```
$ gcc -o prog prog.c func1.o func2.o
```

Si supponga ora di voler includere i file oggetto all'interno di una libreria e di voler utilizzare la libreria per compilare il programma prog. Per fare ciò è possibile utilizzare il programma ar come segue:

```
$ ar q libprog.a func1.o
ar: creating libprog.a
$ ar q libprog.a func2.o
$
```

L'opzione q indica ad ar di appendere il file oggetto alla fine della libreria di nome libprog.a. Dal momento che inizialmente la libreria non esiste, il primo comando crea il file libprog.a e vi inserisce il file oggetto func1.o. Il secondo comando semplicemente aggiunge il file oggetto func2.o alla libreria già esistente.

È possibile verificare il contenuto della libreria col comando

```
$ ar t libprog.a
func1.o
func2.o
```

Ora si può utilizzare la libreria per la compilazione del programma con il comando

```
$ gcc -Wall prog.c ./libprog.a -o prog
```

È possibile inserire in una libreria molti file oggetto. Non tutti devono essere necessariamente linkati nel programma eseguibile, ma soltanto quelli che contengono le funzioni necessarie sono linkati. È da tenere presente però che se in un singolo file oggetto è contenuto il codice di più funzioni, e soltanto un sottoinsieme di queste funzioni sono utilizzate dal programma principale, *tutto il file oggetto prelevato dalla libreria viene linkato nel file eseguibile*, andando ad aumentare le dimensioni del file.

14.2 LA LIBRERIA DI INPUT/OUTPUT STDIO

La libreria di input/output contiene definizioni di macro, costanti, dichiarazioni di funzioni e di tipi utilizzati per le comuni operazioni di input/output.

Per utilizzare le funzioni della libreria di input/output è necessario includere il file di intestazione `stdio.h`, con l'istruzione:

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

La libreria dichiara tutte le funzioni per l'accesso a file, come `fopen`, `fclose`, `fread` e `fwrite`. Data l'importanza dell'accesso a file per i programmi in C, a queste funzioni è stato infatti dedicato l'intero Capitolo 12, e non vengono quindi descritte in questa sezione.

14.2.1 Printf

La funzione di output più comune è

```
int printf(char *, ...);
```

La notazione "..." indica che il numero dei parametri è variabile (si tratta di una funzione variadica).

La funzione `printf` permette di eseguire un output di dati di vari tipi, formattato secondo specifiche definibili in una opportuna stringa di formato che costituisce il primo parametro.

La stringa di formato può essere composta da caratteri ordinari, che vengono copiati sull'output, oppure da specifiche di campo con la seguente forma:

```
%[-][<amp>][.<prec>][l]{d|o|x|u|c|s|e|f|g}
```

dove

- - indica allineamento a sinistra dei dati
- <amp> è l'ampiezza minima del campo
- .<prec> indica la precisione (numero massimo di caratteri di una stringa oppure numero di cifre decimali)
- "l"indica un numero long
- segue un carattere di conversione

Il valore di ritorno di `printf` è il numero di caratteri scritti oppure un numero negativo in caso di errore.

Esempio

```
int i;
float f;

printf("i = %d (%x)\n", i, i);
printf("i = %7.2f\n", f);
```

Il carattere '\n' indica l'andata a capo.

14.3 LA LIBRERIA STANDARD STDLIB

La libreria standard dichiara costanti e funzioni di utilità generale, quali quelle per l'allocazione della memoria, conversione tra tipi e altre.

Per utilizzare le funzioni della libreria standard è necessario includere il file di intestazione `stdlib.h`, con l'istruzione:

```
#include <stdlib.h>
```

Tra i tipi di dato definiti in `stdlib.h` c'è `size_t`, un tipo intero che è il tipo del valore restituito dall'operatore `sizeof`.

Tra le costanti definite in `stdlib.h` c'è la macro `NULL`, generalmente viene definita come 0, 0L, oppure `(void*)0`, che rappresenta il puntatore nullo. Inoltre, la costante `RAND_MAX` (maggiore o uguale a 32767), rappresenta il valore massimo che viene restituito dalla funzione `rand()` (vedi Sezione 14.3.3). Altre funzioni molto utili sono la `qsort` e la `bsearch`, che implementano rispettivamente l'algoritmo di ordinamento Quick Sort e quello per la ricerca binaria su un vettore ordinato. Queste funzioni ed il loro utilizzo verranno descritti in dettaglio in Sezione 19.3.

14.3.1 Controllo dell'esecuzione del programma

Le funzioni più rilevanti per la gestione dell'esecuzione di un programma sono le seguenti:

- `void abort (void);`
  termina immediatamente ed in modo anormale il programma, ed equivale alla ricezione del segnale di abort `SIGABRT`

- `int atexit (void (*func) (void));`
  permette di registrare una funzione la quale sarà eseguita appena prima della normale terminazione del programma; la funzione accetta il puntatore alla funzione da richiamare;

- `void exit (int status);`
  causa la normale terminazione del programma, ritornando il valore `status`;

- `char *getenv (char *name);`
  restituisce la stringa che nell'ambiente di lavoro del programma è associata al nome fornito `name`, oppure `NULL` se non esiste alcuna stringa con quel nome;

- `int system (char *command);`
  passa la stringa fornitale all'ambiente di lavoro per l'esecuzione del comando associato e restituisce il codice d'uscita del comando invocato.

La funzione `atexit` può essere chiamata più volte, per registrare più di una funzione per l'esecuzione al termine del programma. Le funzioni registrate sono eseguite in ordine inverso rispetto alla loro registrazione ed il controllo viene restituito all'ambiente chiamante. Il programma restituisce quindi un valore numerico, che generalmente indica lo stato del programma o la causa della sua terminazione, il quale deve essere fornito alla funzione stessa.

14.3.2 Gestione della memoria

Le funzioni `malloc` e `calloc` allocano dinamicamente blocchi di memoria, la richiesta di nuova memoria viene fatta al sistema operativo che, se possibile, mette a disposizione del programma un nuovo blocco di memoria della dimensione specificata.

Le funzioni inerenti l'allocazione e la gestione della memoria sono le seguenti:

- `void * malloc(size_t n);`
  ritorna un puntatore ad un blocco di memoria di `n` byte
- `void * calloc(size_t n, size_t size);`
  ritorna un puntatore ad un blocco di memoria in grado di contenere un vettore di `n` elementi ciascuno dei quali ha dimensione `size`; il blocco di memoria viene inizializzato a 0
- `void * realloc(void *pt, size_t n);`
  permette di ridimensionare un blocco di memoria già allocato e puntato da `pt`, la nuova dimensione è `n` (il contenuto precedente viene mantenuto).
- `free (void *pt);`
  libera il blocco di memoria di indirizzo `pt` precedentemente allocato tramite `malloc` o `calloc`.

Tutte le funzioni di allocazione restituiscono `NULL` se non è stato possibile soddisfare la richiesta.

14.3.3 Generazione di numeri casuali

Tra le funzioni, ci sono quelle per la generazione di numero casuali (pseudo-casuali, in realtà) che si effettua utilizzando le funzioni `rand` e `srand`. Si parla di numeri pseudo-casuali in quanto la sequenza di numeri generata da `rand` "somiglia" ad una sequenza casuale di numeri, ma in realtà è generata da un opportuno algoritmo, che quindi è un processo deterministico, che tenta di "simulare" il più fedelmente possibile una sequenza casuale[27].

La funzione `rand()` ritorna un numero pseudo-casuale compreso nell'intervallo [0,RAND_MAX]. È semplice ottenere un numero pseudo-casuale in un qualsiasi intervallo [a, b] utilizzando opportunamente l'operatore modulo (vedi Sezione 9.19), come nell'esempio seguente:

```
valore = a + rand() % (b - a + 1);
```

Il generatore di numeri pseudo-casuali può venire inizializzato impostando il valore del *seme* con la funzione `srand`. Il seme determina quale sarà la sequenza di numeri pseudo-casuali fornita dal generatore. L'impostazione del seme si rivela molto utile per ripetere in modo deterministico un programma che utilizza numeri casuali, per esempio per andare alla ricerca di comportamenti particolari che si verificano solo in seguito a determinate sequenze di numeri pseudo-casuali.

Il programma seguente illustra il funzionamento del generatore di numeri pseudo-casuali per simulare il lancio di due dadi.

Il programma è contenuto nel file `dadi.c`.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
```

[27]Il problema di generare numeri casuali con un computer non è banale, e ha una estrema rilevanza in applicazioni come la crittografia, le simulazioni, ecc.

```
4  void lancio(int *a, int *b);
5  void dadi(int n);
6
7  int main(int argc, char **argv)
8  {
9    int seme = 0, n = 1;
10
11   switch (argc) {
12     case 3 :
13       sscanf(argv[2], "%d", &n);
14     case 2 :
15       sscanf(argv[1], "%d", &seme);
16       break;
17     case 1 :
18       break;
19     default : return 1;
20   }
21
22   srand(seme);
23   dadi(n);
24
25   return 0;
26 }
27
28 void lancio(int *a, int *b)
29 {
30   *a = 1 + rand() % 6;
31   *b = 1 + rand() % 6;
32 }
33
34 void dadi(int n)
35 {
36   int a, b, i;
37
38   for (i = 0; i < n; i++) {
39     lancio(&a, &b);
40     printf("%d %d\n", a, b);
41   }
42 }
```

Il programma legge da linea di comando il seme e il numero di lanci da effettuare, rispettivamente in questo ordine. I parametri possono essere entrambi o soltanto uno, in quest'ultimo caso viene inizializzato soltanto il seme. Se i parametri non vengono forniti, sono usati i valori di default, che valgono 0 per il seme e 1 per il numero di lanci. Il generatore di numeri pseudo-casuali viene inizializzato con l'istruzione `srand(seme)`. Si provi a verificare come, fornendo sempre lo stesso seme in ingresso, la sequenza di numeri generata sia sempre la stessa.

Viene poi chiamata la funzione `dadi`, che effettua gli `n` lanci. Per ciascun lancio viene chiamata la funzione `lancio`, la quale attende due parametri interi passati per riferimento; tali parametri conterranno, al termine dell'esecuzione della funzione, i valori dei singoli lanci. L'esito di un lancio viene calcolato mediante la funzione `rand`, limitando opportunamente il valore restituito nell'intervallo $[1 \ldots 6]$ facendo uso dell'operatore modulo.

14.4 MANIPOLAZIONE DI STRINGHE

Una delle librerie più utili disponibili per il linguaggio C è quella per la manipolazione di stringhe e di porzioni di memoria. Queste funzioni sono utilizzabili includendo il file di intestazione `string.h`, con l'istruzione

```
#include <string.h>
```

Le funzioni più importanti ed utilizzate sono le seguenti:

- `void *memcpy (void *dest, void *src, size_t n);`
  copia `n` byte da partire dall'indirizzo `src` in `dest`
- `void *memmove (void *dest, void *src, size_t n);`
  copia `n` bytes dall'indirizzo `src` a `dest`, garantendo il corretto funzionamento nel caso in cui le aree di memoria si sovrappongano
- `void *memset (void *s, int c, size_t n);`
  imposta `n` byte al valore di `c` a partire dall'indirizzo `s`
- `int memcmp (void *s1, void *s2, size_t n);`
  confronta `n` byte a partire dagli indirizzi `s1` e `s2`
- `void *memchr (void *s, int c, size_t n);`
  ricerca il valore `c` negli `n` byte che iniziano all'indirizzo `s`
- `char *strcpy (char * dest, char * src);`
  copia la stringa `src` in `dest`
- `char *strncpy (char *dest, char *src, size_t n);`
  copia la stringa `src` in `dest` fino ad un massimo di `n` caratteri
- `char *strcat (char *dest, char *src);`
  concatena (appende) la stringa `src` in fondo alla stringa `dest`
- `char *strncat (char *dest, char *src, size_t n);`
  concatena (appende) la stringa `src` in fondo alla stringa `dest` fino ad un massimo di `n` caratteri
- `int strcmp (char *s1, char *s2);`
  confronta le due stringhe `s1` e `s2`; ritorna un valore minore di 0 se `s1` è minore di `s2`, 0 se `s1 == s2` e maggiore di 0 se `s1` è maggiore di `s2`
- `int strncmp (char *s1, char *s2, size_t n);`
  confronta `n` caratteri delle stringhe `s1` e `s2`; il valore ritornato è il medesimo di `strcmp`;
- `char *strchr (char *s, int c);`
  trova la prima occorrenza di `c` all'interno della stringa `s`
- `char *strrchr (char *s, int c);`
  trova l'ultima occorrenza di `c` all'interno della stringa `s`
- `char *strstr (char *haystack, char *needle);`
  trova la prima occorrenza della stringa `needle` all'interno della stringa `haystack`
- `char *strtok (char * s, char *delim);`
  divide la stringa `s` in token delimitati dai caratteri in `delim`
- `size_t strlen (char *s);`
  ritorna la lunghezza della stringa `s` in byte

14.5 LA LIBRERIA MATEMATICA

La libreria matematica implementa tutte le funzioni più comuni per effettuare calcoli matematici. Per utilizzare le funzioni della libreria matematica occorre includere il relativo file di intestazione:

```
#include <math.h>
```

Alcune delle funzioni più utilizzate sono:
Funzioni trigonometriche:

- `double acos(double x)`: arcocoseno di x
- `double asin(double x)`: arcoseno di x
- `double atan(double x)`: arcotangente di x
- `double atan2(double x, double y)`: arcotangente di y/x
- `double cos(double x)`: coseno di x
- `double sin(double x)`: seno di x
- `double tan(double x)`: tangente di x

NOTA Tutti gli angoli sono espressi in radianti, non in gradi!

Esponenziali e logaritmi:

- `double exp(double x)`: esponenziale x (e^x)
- `double log(double x)`: logaritmo naturale di x ($\ln x$)
- `double log10(double x)`: logaritmo di x n base 10 ($\log_{10} x$)

Elevamento a potenza:

- `double pow(double x, double y)`: calcola x^y
- `double sqrt(double x)`: calcola $\sqrt{x}$

Arrotondamento:

- `double fmod(double x, double y)`: calcola il modulo x/y
- `double ceil(double x)`: restituisce il più piccolo intero non minore di x
- `double floor(double x)`: restituisce il più grande intero non maggiore di x
- `double fabs(double x)`: restituisce il valore assoluto di x
- `double round(double x)`: arrotonda x all'intero più vicino

La libreria matematica definisce anche una serie di costanti utili nei calcoli matematici più comuni:

```
# define M_E          2.7182818284590452354   /* e */
# define M_LOG2E      1.4426950408889634074   /* log_2 e */
# define M_LOG10E     0.43429448190325182765  /* log_10 e */
# define M_LN2        0.69314718055994530942  /* log_e 2 */
# define M_LN10       2.30258509299404568402  /* log_e 10 */
```

```
# define M_PI          3.14159265358979323846  /* pi */
# define M_PI_2        1.57079632679489661923  /* pi/2 */
# define M_PI_4        0.78539816339744830962  /* pi/4 */
# define M_1_PI        0.31830988618379067154  /* 1/pi */
# define M_2_PI        0.63661977236758134308  /* 2/pi */
# define M_2_SQRTPI    1.12837916709551257390  /* 2/sqrt(pi) */
# define M_SQRT2       1.41421356237309504880  /* sqrt(2) */
# define M_SQRT1_2     0.70710678118654752440  /* 1/sqrt(2) */
```

Capitolo 15

STILE DI PROGRAMMAZIONE

Lo stile di programmazione è un problema che verrà trattato brevemente in questo capitolo. Per "stile di programmazione" si intendono tutte le scelte stilistiche che sono relative al modo di *presentare* il codice sorgente.

A seconda dello stile adottato, un programma scritto in linguaggio C può risultare più o meno comprensibile. Esiste addirittura una manifestazione internazionale, la *International Obfuscated C Code Contest*[28] nella quale l'obiettivo è scrivere programmi C funzionanti che siano più criptici possibile. Il seguente programma è un esempio tratto dal sito web della manifestazione[29]:

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l              ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

[28]IOCCC, `http://www.ioccc.org/`
[29]Disponibile all'indirizzo `http://www.no.ioccc.org/2001/cheong.c.`

Qualunque sia lo stile adottato per la scrittura di un programma, è fondamentale essere coerenti, facendo rientrare i blocchi sempre allo stesso modo, qualunque esso sia. Lo stile più diffuso è quello di Kernighan e Ritchie, che prevede di aprire la parentesi graffa a fine riga e chiudere la parentesi graffa da sola in una riga a se stante. La preferenza stilistica non è comunque ciò che conta, quanto piuttosto la coerenza in tutto il codice sorgente.

15.1 INDENTAZIONE

L'*indentazione* è una tecnica importante per la stesura di un sorgente chiaro. Per indentazione si intende il rientro dei blocchi di codice per mezzo di spazi o tabulazioni inserite ad inizio della riga. Qualunque sia il livello di rientro dei blocchi, 2, 4 o 8 caratteri, il carattere `TAB` vale tipicamente 8 spazi[30].

Senza ricorrere ad esempi "estremi" come quello sopra riportato, è facile intuire come il seguente spezzone di codice male indentato:

```
void funzione() {
  int a,b;
int i,j,condizione;
   for(i=0;i<100;i++) {
 for(j=0;j<100;j++) {
     if(condizione) {
  a=i*j;
     printf("verificata\n");
  } else{
       b=i*j;
  printf("non verificata\n");
     }
  }
}
}
```

sia molto meno comprensibile del seguente:

```
void elabora()
{
  int a, b;
  int i, j, condizione;

  for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
      if (condizione) {
        a = i * j;
        printf("verificata\n");
      } else {
        b = i * j;
        printf("non verificata\n");
      }
    }
  }
}
```

[30] Attenzione alla configurazione predefinita dell' editor che potrebbe essere diversa.

Dov'è la differenza?

Si nota subito che nel secondo esempio le istruzioni che fanno parte di un determinato blocco sono rientrate rispetto all'istruzione che inizia il blocco e all'eventuale parentesi di chiusura del blocco. Per esempio, il blocco

```
for (j = 0; j < 100; j++) {
  if (condizione) {
    a = i * j;
    printf("verificata\n");
  } else {
    b = i * j;
    printf("non verificata\n");
  }
}
```

costituisce il corpo del bloccoi terativo iniziato dal primo `for`, mentre si vede chiaramente che

```
b = i * j;
printf("non verificata\n");
```

è il corpo della `else`.

15.2 INDENTAZIONE ED ERRORI

Un errore tipico che si commette imparando a programmare, è quello di dimenticare di chiudere le parentesi precedentemente aperte. Anche programmatori esperti possono commettere questo tipo di errori, ma l'esperienza aiuta a risolvere molto velocemente questi errori, in particolare se generano errori in compilazione.

```
int main()
{

void elabora()
{
  int a, b;
  int i, j, condizione;

  for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
      if (condizione)
        a = i * j;
      printf("verificata\n");
```

Il codice precedente è stato scritto "di fila", con una chiara indentazione. Si è proceduto ad aprire parentesi ed a scrivere il corpo dei vari blocchi. Ora è difficile chiudere correttamente le parentesi, mentre è facile commettere errori sintattici. Un errore macroscopico che si nota subito, è il fatto che il codice scritto com'è, anche se si chiudessero correttamente le parentesi relative alla funzione `elabora`, presenta una parentesi aperta che introduce il corpo della funzione `main` e che non viene chiusa prima di iniziare la definizione di `elabora`. Questo produce inevitabilmente un errore in compilazione.

Un utile stratagemma per evitare di dimenticare la chiusura delle parentesi è quello di scrivere subito, ogni volta che c'è un blocco di codice delimitato da parentesi, sia la parentesi di apertura che quella di chiusura, e poi scrivere il codice all'interno del blocco. In questo modo si può essere ragionevolmente certi che ad ogni parentesi aperta ne corrisponde una chiusa. Per esempio:

```
int main()
{
  /* aperta e chiusa la parentesi si puo' inserire il codice */
}
```

oppure

```
void elabora()
{
  int a, b;
  int i, j, condizione;

  for (i = 0; i < 100; i++) {
    /* aperta e chiusa la parentesi si puo' inserire il codice */
  }
}
```

Da notare che anche i commenti sono indentati opportunamente, per far intendere a quale blocco di codice fanno riferimento.

Questo suggerimento non vale soltanto per i blocchi di codice delimitati da parentesi graffe, ma anche quelli delimitati da parentesi tonde in condizioni o espressioni complesse. Nell'esempio seguente:

```
  if ((a != 3) && ((b < 100) || (  /* espressione */ )))
```

può non essere banale inserire tutte le espressioni con la corretta sequenza di parentesi. Allora, ogni volta che viene aggiunta un'espressione, si aprono e chiudono le rispettive parentesi, e poi si inserisce l'espressione.

15.3 INDICAZIONI VARIE

Le funzioni vanno mantenute brevi e comprensibili. Se una funzione diventa troppo complessa è meglio dividerne il codice in blocchi concettualmente separati, implementandoli sotto forma di funzioni distinte.

L'utilizzo delle strutture dati migliora la chiarezza e la manutenibilità del programma. Per gestire le strutture dati, è generalmente meglio definire delle funzioni per l'inizializzazione e la distruzione degli oggetti invece che usare variabili globali. Ciò significa che è preferibile implementare delle funzioni dedicate all'inizializzazione dei campi di una struttura, e all'eventuale allocazione/deallocazione della memoria relativa, invece che dichiarare variabili globali allocate staticamente.

È bene controllare e gestire opportunamente sempre tutti gli errori: ogni funzione che viene chiamata può fallire, e quindi il codice chiamante deve verificare il valore di ritorno e comportarsi in maniera appropriata, che spesso vuol dire propagare l'errore alla funzione chiamante.

La chiamata alla funzione `exit` dall'interno di una funzione in caso di errore è da sconsigliare, ed è meglio lasciare decidere al programma principale come gestire l'errore.

Commentare bene il codice: in linea di massima, costrutti particolarmente *furbi* sono da evitare, in quanto possono essere difficili da interpretare da altri programmatori o dallo stesso programmatore a distanza di tempo. Se tali costrutti vengono utilizzati, è bene spiegare il perché di tale scelta.

Specificare sempre nel file sorgente i termini di licenza con la quale il programma viene rilasciato. In assenza di permessi specifici vale la clausola *tutti i diritti riservati*, ma anche se questa è la vostra intenzione è sempre meglio specificarlo per chiarire ogni dubbio.

Non fare interazione utente se non strettamente necessario. Se necessario, leggere `stdin` con `fgets` e poi usare `sscanf` per interpretare i dati in ingresso, mai con `scanf` direttamente. Scrivere `stdout` per righe complete, terminate da '\n'. Evitare l'output inutile (*il silenzio è d'oro*) e le righe

vuote superflue. Alcuni di questi suggerimenti non sono stati applicati in taluni esempi presentati, ma ciò è stato fatto tipicamente perché ne derivava un codice più semplice e leggibile, quindi più adatto agli scopi didattici del documento.

Una nota particolare è richiesta dalla raccomandazione di non usare `scanf` per leggere eventuali dati di input necessari all'elaborazione. Nel corso di questo documento è stata utilizzata varie volte tale funzione per leggere dati da tastiera e poi procedere con la computazione. Questo è stato fatto per motivi didattici, in quanto si ritiene che l'uso di `fgets` (vedi Sezione 12.6) richieda dei concetti troppo avanzati per essere introdotti nei primi esempi presentati.

Capitolo 16

TECNICHE DI PROGRAMMAZIONE

Alcune tecniche di programmazione che saranno presentate in questo capitolo sono alla base di molte tecniche per la gestione delle informazioni. Sostanzialmente ci si concentrerà sulla *ricorsione*, illustrata di seguito.

16.1 LA RICORSIONE

La *ricorsione* è una tecnica di programmazione che viene realizzata facendo in modo che una funzione richiami sè stessa. La funzione che richiama sè stessa è detta *funzione ricorsiva*.

La ricorsione è una tecnica molto utile poiché permette di implementare in modo chiaro ed elegante molti algoritmi che si prestano ad un tipo di implementazione ricorsiva. Molti di tali algoritmi sono spesso esplicitamente definiti in termini ricorsivi. Alcuni esempi sono:

- la ricerca in un albero o in un grafo;
- la generazione di forme regolari (frattali, insiemi di Mandelbrot, ecc.);
- il calcolo di funzioni (es. il fattoriale, ecc.);
- la generazione di successioni di numeri (es. i numeri di Fibonacci, ecc.).

Come esempio si consideri il seguente programma che calcola l'i-esimo numero di Fibonacci. Il valore di i viene letto dalla linea di comando.

Come noto, i numeri di Fibonacci si ottengono sommando i due numeri precedenti nella sequenza. Matematicamente parlando, assegnati i primi due valori della successione $a_0 = 0$ e $a_1 = 1$, si ha che

$$a_i = a_{i-2} + a_{i-1}, \quad i \geq 2$$

La sequenza dei primi 20 numeri numeri è dunque

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

e il calcolo dell'i-esimo numero della serie può essere effettato richiamando ricorsivamente la funzione che somma i due numeri precedenti della successione.

Il programma è contenuto nel file `fibonacci.c`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  long fibonacci(long n)
5  {
6    if (n <= 1) return n;
7    return fibonacci(n - 1) + fibonacci(n - 2);
8  }
9
10 int main(int argc, char **argv)
11 {
12   long n;
13
14   if (argc != 2) return 1;
15
16   if (n = atoi(argv[1]))
17     printf("fibonacci(%d) = %d\n", n, fibonacci(n));
18
19   return 0;
20 }
```

Un aspetto importante implicito nell'uso delle funzioni ricorsive è che la funzione deve prevedere una condizione di terminazione della ricorsione, ovvero una condizione per la quale la funzione ritorna senza chiamare sè stessa. Nell'esempio dei numeri di Fibonacci, la funzione ritorna quando il numero passato come argomento è minore o uguale a 1 (linea 14).

Se da un lato la ricorsione ha il vantaggio della semplicità e chiarezza di implementazione, dall'altro comporta qualche inconveniente. Il problema principale consiste nell'occupazione dello stack, in quanto ad ogni chiamata alla funzione ricorsiva le variabili definite localmente alla funzione devono essere allocate sullo stack, insieme alle informazioni per il rientro dalla funzione chiamata. Questo può provocare problemi di occupazione della memoria in caso di una sequenza di ricorsione molto lunga. Inoltre, a causa delle continue chiamate a funzione, la ricorsione è più svantaggiosa anche in termini di tempo rispetto, ad esempio, ad un ciclo classico realizzato con i costrutti `for` o `while`.

Infine, è da sottolineare che tutti gli algoritmi implementabili in forma ricorsiva sono anche implementabili in forma iterativa. Spesso però nel caso di implementazione in forma iterativa, l'implementazione stessa risulta meno elegante e comprensibile. Il programma che genera i numeri di Fibonacci nella versione non ricorsiva è la seguente:

Il programma è contenuto nel file `fibonacci_rev.c`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N (20)
5
6  int main()
7  {
8    int num[N];
9    int i;
10
11   num[0] = 0;
12   num[1] = 1;
13
14   for (i = 2; i < N; i++) {
15     num[i] = num[i - 1] + num[i - 2];
16   }
17
18   printf("fibonacci-rev:");
19   for (i = N - 1; i >= 0; i--) {
20     printf(" %d", num[i]);
21   }
22   printf("\n");
23
24   return 0;
25 }
```

Questa versione del programma è leggermente diversa da quella precedente, in quanto stampa i numeri della successione in ordine inverso (ciclo alla linea 19). Il punto sul quale focalizzarsi, comunque, è il ciclo `for` alla linea 14 che calcola i numeri della successione. Si tratta quindi di un costrutto iterativo invece che di una ricorsione.

Capitolo 17

STRUTTURE INFORMATIVE

In questo capitolo verranno descritte le strutture informative più comuni che vengono utilizzate come "mattoni elementari" per costruire algoritmi complessi e dedicati a specifiche attività. In particolare, questo capitolo presenterà le cosiddette strutture astratte di dati, mentre il prossimo capitolo sarà dedicato alle strutture concrete che implementano le strutture astratte.

Capita sovente che tali tecniche vengano presentate senza portare esempi concreti del loro utilizzo in situazioni pratiche, e quindi sono spesso percepite come fini a loro stesse. Nelle sezioni seguenti, ciascuna dedicata ad una specifica tecnica di programmazione o algoritmo, si avrà cura di riportare esempi di casi concreti nei quali tali tecniche trovano impiego.

17.1 CLASSIFICAZIONE DELLE STRUTTURE DI DATI

L'informazione elaborata da un calcolatore si presenta in una varietà di forme differenti che dipendono in genere dalla natura del problema da risolvere. Accanto a problemi che richiedono il trattamento di dati propriamente numerici, ve ne sono altri di natura non numerica che richiedono elaborazioni su sequenze di caratteri alfabetici, su schemi di flusso o grafi e altre opportune rappresentazioni.

I dati si presentano quindi strutturati in modi differenti, per ciascuno dei quali è necessario individuare una rappresentazione interna al calcolatore che risulti conveniente per le elaborazioni da eseguire e per lo scambio di informazioni con l'esterno.

Con il termine strutture informative, si comprendono:

- le *strutture dati astratte*, proprie del problema e dipendenti unicamente da questo; tali strutture sono definite da un insieme di leggi che stabiliscono le relazioni esistenti fra i dati di un insieme finito;

- le *strutture dati concrete*, ovvero relative allo stato interno della memoria nella quale le strutture sono memorizzate; le strutture concrete sono individuate dall'insieme di celle contenenti le informazioni e gruppi di regole per il loro ordinamento logico.

Dal momento che ciascuna applicazione è spesso studiata per utilizzare specifiche strutture di dati, è opportuno esaminare i principali tipi di aggregati di dati dotati di una struttura logica, cioè le strutture astratte di dati, e i sistemi per la loro rappresentazione nella memoria di un calcolatore, cioè le possibili strutture concrete adatte a contenere le strutture astratte.

La formulazione di un problema sarà espressa tenendo conto anche del tipo di rappresentazione dei dati in memoria che si pensa di adottare. Infatti, la scelta delle strutture concrete per la memorizzazione delle informazioni può avere un impatto notevole su cifre di merito come l'efficienza di calcolo e l'occupazione di memoria, che sono i tipici parametri che si desidera ottimizzare nell'implementazione di un algoritmo.

17.2 STRUTTURE ASTRATTE DI DATI

Una struttura dati astratta consiste in un insieme finito di dati nel quale è definita una legge di ordinamento, cioè è stabilita una corrispondenza biunivoca tra i suoi elementi e l'insieme dei primi n numeri naturali.

In base a tale legge, è possibile stabilire:

- qual è il primo elemento dell'insieme;
- qual è l'ultimo elemento dell'insieme;
- quale di due elementi qualsiasi precede l'altro.

Le strutture dati astratte che verranno prese in considerazione sono:

- la lista lineare;
- la coda;
- la pila, o stack;
- la doppia coda;
- l'array, nella forma di vettore e matrice;
- la tavola;
- il grafo;
- l'albero.

Le strutture dati sopra elencate vengono impiegate come "mattoni" per realizzare complessi algoritmi e sistemi di calcolo. È utile conoscere, almeno in modo orientativo, quali sono gli impieghi tipici delle singole strutture dati. Per questo di seguito vengono elencate alcune delle tipiche applicazioni delle strutture considerate.

- la lista lineare si usa per
  - memorizzare matrici sparse di grandi dimensioni (aventi molti zeri e pochi elementi diversi da zero)
  - gestire i blocchi liberi/occupati della memoria di un calcolatore

- la coda si utilizza per
  - schedulare l'esecuzione di attività di calcolo in base al tempo di arrivo delle richieste di esecuzione in un sistema operativo
- la pila, o stack
  - serve a memorizzare i dati locali e gli indirizzi di ritorno delle subroutine
  - serve a valutare espressioni scritte in forma polacca inversa
- l'array, nella forma di vettore, si usa per
  - contenere i dati per calcoli matematici
  - memorizzare i campioni sonori per realizzare un buffer in applicazioni di elaborazione audio
- l'array, nella forma di matrice, viene impiegato per
  - contenere i dati per calcoli matematici
  - memorizzare e manipolare immagini (le tipiche trasformazioni grafiche sono basate su calcoli matriciali)
  - applicazioni di controllo automatico
- le tavole
  - insieme a strutture a grafo, sono alla base delle basi di dati di tipo relazionale, uno dei modelli più diffusi e utilizzati
- il grafo viene utilizzato per
  - rappresentare i collegamenti nelle reti di agenti (calcolatori, robot, ecc.)
  - rappresentare i legami tra gli elementi di un insieme
  - nella navigazione automatica (es. navigazione robotica; nei navigatori basati su mappe, per descrivere le vie di comunicazione stradale)
- l'albero
  - viene usato per mantenere elenchi ordinati di elementi (alberi binari)
  - essendo un grafo, viene anch'esso usato per raggruppare gli elementi che fanno capo a entità comuni (es. individuazione di oggetti distinti in una immagine mediante segmentazione e algoritmi tipo "sparse forest")

17.3 LA LISTA LINEARE

La *lista lineare* è una successione di elementi omogenei disposti in memoria in posizione qualsiasi. Ciascun elemento contiene una informazione e un puntatore all'elemento successivo. Gli elementi della lista devono essere omogenei fra loro. Per accedere ad una lista è necessario soltanto l'indirizzo del suo primo elemento, pertanto nelle liste l'accesso ad un elemento deve necessariamente avvenire tramite una ricerca sequenziale a partire dal primo elemento della lista. Nel caso in cui gli elementi della lista siano i caratteri di un alfabeto, si parla di *stringa*.

Le operazioni sulle liste sono di due tipi. Le *operazioni globali* riguardano tutti gli elementi della lista, mentre le *operazioni locali* riguardano i singoli elementi della lista.

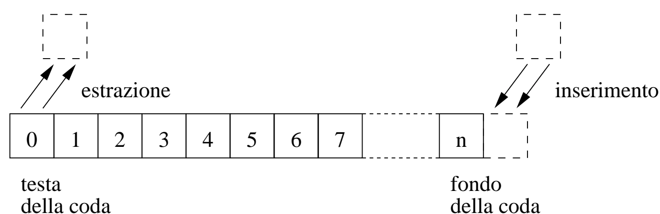


Figura 17.1 Le operazioni che si possono attuare su una coda.

Esempi di operazioni globali sono:

- concatenazione o fusione di due liste in una sola
- suddivisione di una lista in più parti
- ordinamento degli elementi secondo un criterio diverso da quello iniziale

Esempi di operazioni locali sono:

- lettura e/o modifica di un elemento della lista
- inserimento di un nuovo elemento nella lista
- eliminazione di un elemento della lista

La nozione di lista è generalizzabile, ovvero gli elementi della lista possono, a loro volta, essere delle liste i cui elementi possono essere ancora delle liste e così di seguito. Da notare che le liste sono strutture dinamiche, in quanto la loro dimensione può variare durante il loro utilizzo.

17.4 LA CODA

La *coda* è un particolare tipo di lista lineare di lunghezza variabile nella quale:

1. gli inserimenti avvengono solo dopo l'ultimo elemento, cioè dal cosiddetto *fondo della coda*
2. le eliminazioni avvengono solo dal primo elemento, ovvero dalla *testa della coda*

Il primo elemento che può essere estratto è il primo ad essere stato inserito. La logica di funzionamento della coda è ben espressa dall'acronimo inglese *FIFO*, che significa *First In First Out*. Per l'appunto, questo indica che il "piu' vecchio" dato inserito nella coda è quello che può essere estratto, come rappresentato in Figura 17.1.

17.5 LA PILA (STACK)

La *pila*, o *stack* è un tipo particolare di lista lineare avente lunghezza variabile in cui gli inserimenti e le estrazioni avvengono ad un solo estremo, cioè dal cosiddetto *fondo della pila*.

La pila è una struttura dati che viene gestita con una modalità di accesso di tipo *Last In First Out* (LIFO). Ciò significa che il dato che può essere prelevato, o letto, da uno stack è soltanto l'ultimo dato che è stato inserito. Le operazioni di inserimento (*push*) e di estrazione (*pop*) sono schematizzate in Figura 17.2.

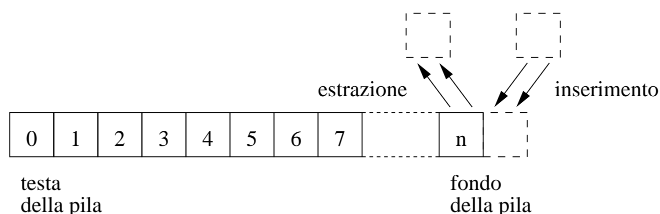


Figura 17.2 Push e pop: le principali operazioni che si compiono su uno stack.

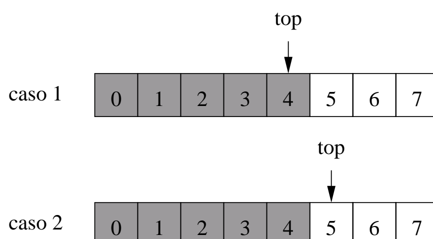


Figura 17.3 Due differenti implementazioni di uno stack.

Il funzionamento della pila richiama l'impilamento di oggetti qualsiasi, per esempio dei piatti. L'impilamento, o inserimento, di un piatto viene fatto in cima alla pila. Viceversa, è possibile prelevare dalla pila soltanto il piatto che si trova in cima alla pila.

La pila ha un gran numero di utilizzi in campo informatico, tra i quali:

- serve a memorizzare i dati locali e gli indirizzi di ritorno delle subroutine
- serve a valutare espressioni scritte in forma polacca inversa

L'implementazione di una pila può essere fatta in vari modi. La più semplice implementazione si basa su un vettore e su un indice che tiene traccia dell'ultimo elemento inserito, come evidenziato nel caso 1 in Figura 17.3. In alternativa, l'indice può indicare il primo elemento libero in cima alla pila, corrispondente al caso 2 in Figura 17.3. La differenza tra le due soluzioni risiede nell'ordine con cui si inseriscono e prelevano gli elementi e quello con cui viene aggiornato l'indice.

Di seguito viene illustrata l'implementazione completa di uno stack di valori interi realizzata per mezzo di un vettore. Potrebbe sembrare limitativa la scelta di memorizzare dei semplici numeri interi, ma non lo è. Per memorizzare dati aventi una struttura ed un significato generico nello stack illustrato, si potrebbero infatti comodamente memorizzare nello stack dei puntatori, che verrebbero maniporati come valori interi. A questo punto il puntatore potrebbe puntare a generiche strutture dati aventi il formato adatto per memorizzare qualsiasi informazione risulti necessaria.

L'eleganza del programma risiede nell'immagazzinare tutti i dati che servono per la gestione dello stack all'interno di una struttura dati, la struttura `struct t_stack`. Tutte le funzioni che operano sullo stack riceveranno come parametro una di queste strutture (in realtà, un puntatore alla struttura), e agiranno sui dati contenuti leggendone e/o modificandone il valore se necessario.

Questo approccio, che è molto comune in programmi di grosse dimensioni, consente due importanti vantaggi:

1. permette di mantenere variabili logicamente collegate tra loro in una struttura che è fisicamente unitaria;

2. permette di gestire più stack diversi in un solo programma con la semplice dichiarazione di tante variabili di tipo `struct t_stack` quanti sono gli stack da gestire.

I dati memorizzati nella struttura sono:

- l'indice `top` del primo elemento libero nello stack;
- `max`, ovvero il numero massimo di elementi immagazzinabili nello stack;
- il puntatore che conterrà l'indirizzo dell'area di memoria per la memorizzazione del vettore.

Il programma è contenuto nel file `stack_vettore.c`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct t_stack {
5    int top;
6    int max;
7    int *stack_arr;
8  };
9
10 int scelta();
11 int init(struct t_stack *c, int max);
12 int dealloc(struct t_stack *c);
13 int push(struct t_stack *c, int val);
14 int pop(struct t_stack *c, int *val);
15 void display(struct t_stack *c);
```

Le funzioni che vengono dichiarate ed utilizzate nel programma sono le seguenti:

- `scelta` richiede all'utente quale operazione intende compiere sullo stack; l'interazione utente è presente al solo scopo di permettere la semplice verifica del funzionamento del programma;
- `init` inizializza un nuovo stack;
- `dealloc` rilascia tutte le risorse allocate dinamicamente;
- `push` inserisce l'elemento `val` nello stack, e ritorna un codice che indica se l'operazione ha avuto successo;
- `init` preleva l'elemeto `val` dallo stack, passato per riferimento, ritornando un codice che indica se l'operazione ha avuto successo;;
- `display` visualizza il contenuto dello stack.

La funzione `main` dichiara la variabile `stack` di tipo `struct t_stack`. Si noti come il campo puntatore della struttura sia inizializzato a `NULL`, per indicare che non è ancora stato allocato spazio. Successivamente viene richiamata la funzione `init` per inizializzare uno stack di 2 elementi, ed effettua un ciclo infinito che continua a richiedere all'utente quale operazione intende compiere sullo stack. La scelta dell'utente viene interpretata mediante il costrutto `switch`. Le operazioni possibili sono l'inserimento e il prelevamento di valori, la stampa dello stack e l'uscita dal programma previa deallocazione della memoria allocata dinamicamente.

```
1  int main()
2  {
3    struct t_stack stack = {.stack_arr = NULL};
4    int sc;
5    int val, ret;
6
7    init(&stack, 2);
8
9    while(1) {
10     sc = scelta();
11
12     switch (sc) {
13       case 1 :
14         printf("elemento da inserire: ");
15         scanf("%d", &val);
16         ret = push(&stack, val);
17         if (ret)
18           printf("** stack overflow: impossibile inserire il dato\n");
19         break;
20       case 2:
21         ret = pop(&stack, &val);
22         if (ret)
23           printf("** stack underflow: nessun elemento da estrarre\n");
24         else printf("elemento estratto : %d\n", val);
25         break;
26       case 3:
27         display(&stack);
28         break;
29       case 4:
30         dealloc(&stack);
31         exit(1);
32       default:
33         printf("Scelta errata\n");
34     }   /* fine switch */
35   }   /* fine while */
36 }   /* fine main() */
```

La funzione `scelta` si limita a presentare un piccolo menu all'utente e a leggere il valore numerico corrispondente alla scelta. Non vengono effettuati controlli particolari: questa funzione potrebbe complicarsi a piacere per gestire eventuali input errati.

```
1  int scelta()
2  {
3    int sc;
4
5    printf(" 1) Push\n");
6    printf(" 2) Pop\n");
7    printf(" 3) Stampa\n");
8    printf(" 4) Esci\n");
9    printf("scelta: ");
10   scanf("%d", &sc);
11
```

```
12    return sc;
13  }
```

La funzione `init` inizializza uno stack di `max` elementi descritto dalla struttura puntata da `c`. La funzione ha successo soltanto se lo stack conterrà almeno un elemento. Il vettore di valori interi viene allocato con la `malloc`. Infine, il valore di `max` viene memorizzato nella struttura e `top` viene inizializzato al valore -1, a significare che lo stack non contiene ancora nessun valore valido. In questo modo si intuisce come il valore di `top` indicherà l'indice dell'ultimo elemento inserito, come diverrà più chiaro analizzando le funzioni di inserimento e prelevamento.

```
1   int init(struct t_stack *c, int max)
2   {
3     if (max <= 0) return 1;   /* almeno un elemento... */
4
5     c->stack_arr = malloc(max * sizeof(int));
6     if (!c->stack_arr) return 2;
7
8     c->top = -1;
9     c->max = max;
10
11    return 0;
12  }
```

La funzione `dealloc` rilascia la memoria precedentemente allocata per il vettore di valori, controllando preventivamente che il puntatore sia non-nullo, in quanto la chiamata della `free` su un puntatore non allocato può causare grossi problemi.

```
1   int dealloc(struct t_stack *c)
2   {
3     if (c->stack_arr)
4       free(c->stack_arr);
5
6     return 0;
7   }
```

La funzione `push` inserisce il valore `val` nello stack identificato dalla struttura puntata da `c`. Il valore viene inserito soltanto se lo stack non è già pieno, ovvero `top` è minore della dimensione dello stack meno uno, cioè c'è spazio per inserire l'elemento richiesto. L'inserimento viene effettuato incrementando il valore di `top` e scrivendo il valore dal memorizzare all'indice così calcolato.

```
1   int push(struct t_stack *c, int val)
2   {
3     if (c->top == (c->max - 1))
4       return 1;
5     else {
6       c->top++;
7       c->stack_arr[c->top] = val;
8       return 0;
9     }
10  }
```

Per estrarre l'ultimo elemento inserito si usa la funzione `pop`, la quale ritorna un codice di errore non nullo se lo stack è vuoto. Altrimenti viene assegnato al parametro passato per indirizzo il valore attualmente memorizzato all'indice `top`; dopodiché l'indice viene decrementato.

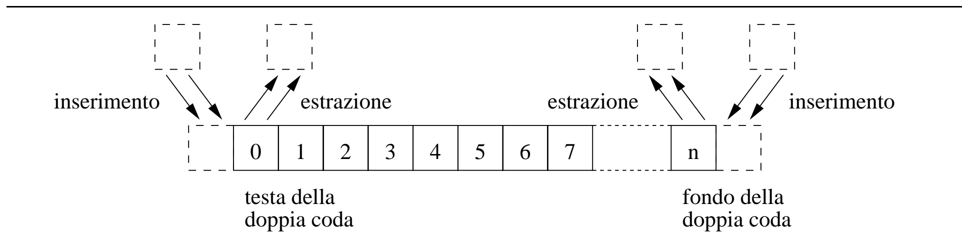


Figura 17.4 Le operazioni che si possono attuare su una doppia coda.

```
1  int pop(struct t_stack *c, int *val)
2  {
3    if (c->top == -1)
4      return 1;
5    else {
6      *val = c->stack_arr[c->top];
7      c->top--;
8      return 0;
9    }
10 }
```

Infine, la funzione `display` visualizza il contenuto dello stack, stampando a video tutti i valori contenuti nel vettore.

```
1  void display(struct t_stack *c)
2  {
3    int i;
4
5    if (c->top == -1)
6      printf("** lo stack e' vuoto\n");
7    else {
8      printf("** elementi dello stack:\n");
9      for(i = c->top; i >=0; i--)
10       printf("  %d\n", c->stack_arr[i]);
11   }
12 }
```

17.6 LA DOPPIA CODA

La *doppia coda* è un tipo di lista lineare a lunghezza variabile in cui gli inserimenti e le estrazioni possono avvenire indifferentemente su entrambi gli estremi, come schematizzato in Figura 17.4.

17.7 GLI ARRAY

Si tratta di un insieme finito di elementi in corrispondenza biunivoca con un insieme di n-ple di numeri interi, detti *indici*. Gli indici possono assumere valori compresi in un intervallo determinato. Inoltre

- per $n = 1$, si parla di *vettore*, o array monodimensionale;

- per $n = 2$, si parla di *matrice*, o array bidimensionale;
- per $n > 2$, si parla di array multi-dimensionale.

L'array è una struttura a lunghezza fissa in cui l'accesso ad un elemento avviene attraverso la n-pla di indici e non in modo sequenziale come avviene nelle liste.

Un vettore si distingue quindi da una lista lineare per il fatto che l'accesso all'elemento di indice i avviene direttamente attraverso l'indice i, mentre l'accesso ad un elemento della lista avviene tramite una ricerca sequenziale che esamina tutti gli elementi della lista fino al reperimento dell'elemento voluto.

17.8 LE TAVOLE (TABELLE)

La *tavola* o *tabella* è un insieme finito di elementi, ciascuno dei quali costituito da una coppia ordinata di dati. Il primo elemento è detto *nome* o *chiave* dell'elemento; il secondo elemento è detto *valore* ed è costituito da informazioni associate alla chiave. L'accesso ad un elemento della tavola avviene tramite la chiave. Le tavole sono utilizzate quando esistono corrispondenze biunivoche tra insiemi non esprimibili tramite formule matematiche

17.9 I GRAFI

Il *grafo* è una struttura dati costituita da:

- un insieme finito di punti detti *nodi* o *vertici*;
- un insieme finito di segmenti, detti *lati* o *archi*, che congiungono coppie di nodi; gli archi possono essere convenientemente identificati dai nomi delle coppie di nodi da essi congiunti.

Se ogni nodo viene considerato il supporto di un dato e i lati come la rappresentazione di una relazione tra i dati contenuti nei nodi da essi uniti, allora il grafo può essere visto come la rappresentazione di una struttura astratta di dati. Alcune tipologie di grafo sono:

- i *grafi connessi* in cui, scelta una coppia qualsiasi di nodi, è sempre possibile congiungere tali nodi mediante un cammino (vedi Figura 17.5);
- i *grafi orientati*, nei quali i lati del grafo sono orientati, e spesso rappresentati graficamente mediante archi che terminano con una freccia (vedi Figura 17.6).

Alcune definizioni relative ai grafi sono:

- due nodi si dicono *adiacenti* se esiste un arco che li congiunge
- un *cammino* o *percorso* è una successione di nodi adiacenti. In particolare si distinguono:
  - un *cammino semplice* è una successione di nodi distinti, ad eccezione eventualmente del primo e dell'ultimo che possono coincidere
  - un *ciclo* o *circuito* è un cammino semplice che congiunge un nodo con sè stesso

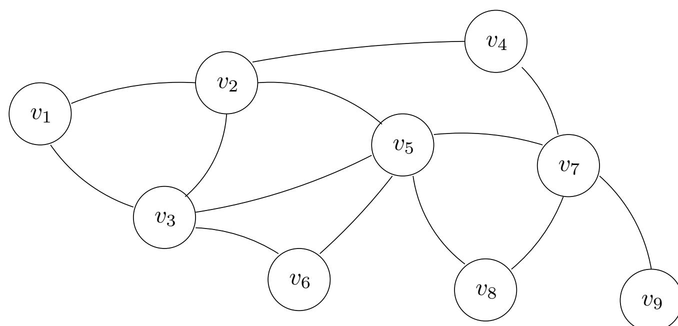


Figura 17.5 Esempio di grafo connesso composto da 9 nodi.

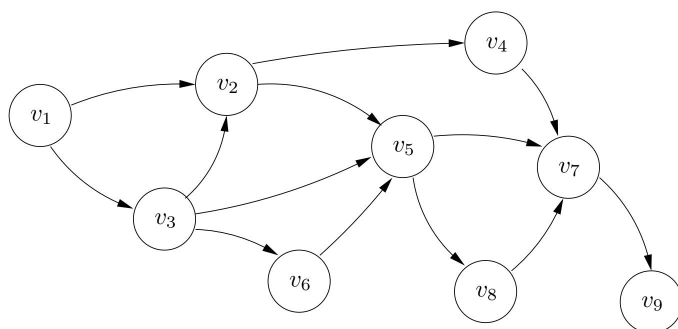


Figura 17.6 Esempio di grafo diretto composto da 9 nodi.

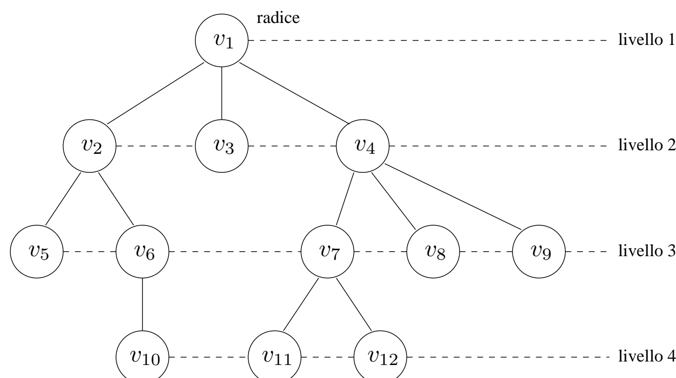


Figura 17.7 Esempio di albero.

17.10 GLI ALBERI

Un *albero libero* è un grafo connesso privo di cicli. Per un albero costituito da n nodi, valgono le seguenti proprietà:

- l'albero contiene $n - 1$ archi
- esiste un solo percorso semplice tra ogni coppia di nodi dell'albero
- se si rimuove un arco qualsiasi dell'albero, la struttura risultante non è più connessa, ma composta da due alberi distinti

Assegnato un albero, scelto un nodo arbitrario come *radice*, si possono ordinare i suoi nodi in base al relativo *livello*:

- il livello della radice è 1
- il livello di ogni altro nodo è pari al numero di nodi contenuti nel percorso tra quel nodo e la radice

Nota la radice, è anche nota la suddivisione in livelli.

Un esempio di albero è riportato in Figura 17.7. La radice dell'albero è il nodo v_1.

Con il termine *foglia* si indicano quei nodi che non appaiono in alcun percorso semplice fra un altro nodo e la radice. In Figura 17.7 le foglie sono costituite dai nodi v_3, v_5, v_8, v_9, v_{10}, v_{11}, v_{12}.

Le strutture informative assumono spesso la forma di alberi in cui sia stabilita la radice, e sono detti semplicemente *alberi*. L'albero si dice *ordinato* se in ciascun livello si considera significativo l'ordine con cui compaiono i nodi. Le proprietà degli alberi liberi valgono anche per gli alberi.

Per gli alberi, si può dare una definizione che non fa riferimento agli archi. Un albero è un insieme costituito da uno o più nodi tale che:

- un particolare nodo è designato come radice
- i rimanenti nodi, se esistono, possono essere suddivisi in insiemi disgiunti, ciascuno dei quali è a sua volta un albero detto *sottoalbero* (si noti che ciascun sottoalbero ha una propria radice).

Questa definizione è ricorsiva poiché espressa in funzione di altri alberi.

In Figura 17.7 sono presenti diversi sottoalberi. Tra gli altri, vi sono quelli che hanno come radice i nodi v_2, v_4, v_6, v_7.

Il *grado* di un nodo è il numero i sottoalberi del nodo stesso.

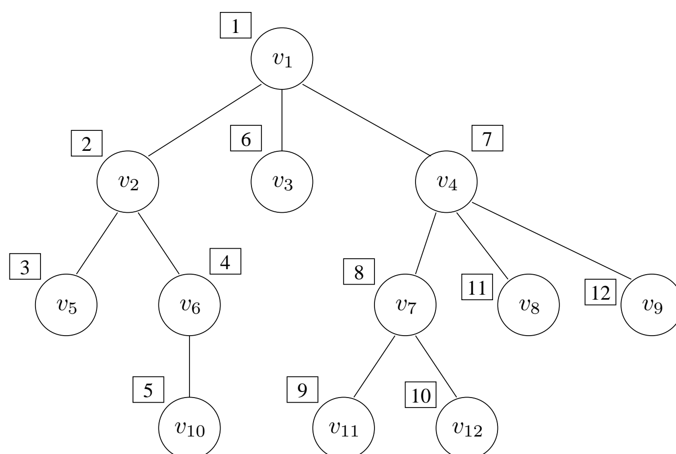


Figura 17.8 Visita anticipata dell'albero di Figura 17.7.

17.10.1 Visita degli alberi

La *visita* di un albero consiste nell'esaminare tutti i suoi nodi uno per uno in ordine appropriato. La soluzione più banale è quella di ripartire ogni volta dalla radice, dopo aver esaminato un nodo; tale soluzione è fortemente inefficiente.

Sono stati proposti metodi di visita di un albero che prevedono un ordinamento particolarmente efficiente per alberi ordinati. I metodi di visita in ordine anticipato e in ordine differito si distinguono per il tempo di esame di ogni nodo rispetto ai suoi sottoalberi. Questi metodi si basano su sequenze di azioni di natura ricorsiva.

17.10.2 Visita in ordine anticipato

La visita in *ordine anticipato* prevede le seguenti azioni:

1. esamina la radice
2. sia $n \geq 0$ è il grado (numero di sottoalberi) della radice; allora
   - visita il primo sottoalbero, in ordine anticipato;
   - visita il secondo sottoalbero, in ordine anticipato;
   - ...
   - visita l'n-esimo sottoalbero, in ordine anticipato

La visita di ciascun sottoalbero avviene partendo dalla radice del sottoalbero stesso, che è il nodo collegato alla radice del passo precedente. Si noti che in questo caso la radice viene esaminata *prima* di esaminare i relativi sottoalberi.

In Figura 17.8 è riportata la sequenza determinata dalla visita anticipata dell'albero di Figura 17.7. I numeri all'interno dei box rettangolari rappresentano la sequenza di visita dei corrispondenti nodi. Un modo alternativo di rappresentare la sequenza di visita è il seguente:

$$v_1 \quad [[v_2 \quad ((v_5) \quad (v_6 \quad (v_{10})))] \quad [v_3] \quad [v_4 \quad (v_7 \quad ((v_{11}) \quad (v_{12})) \quad (v_8) \quad (v_9))]]$$

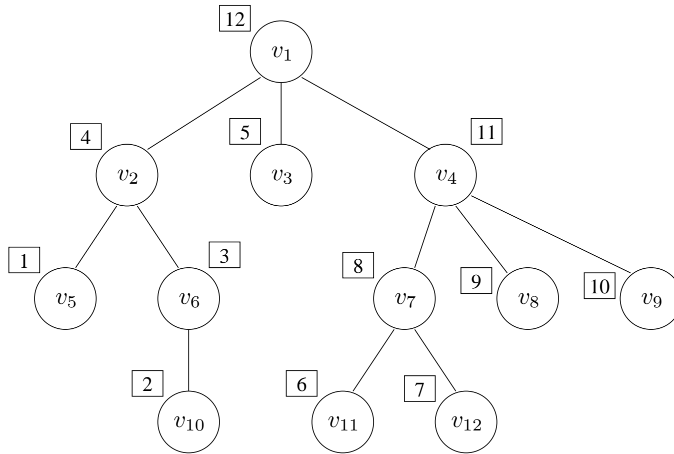


Figura 17.9 Visita differita dell'albero di Figura 17.7.

dove le parentesi sono utilizzate per evidenziare i sottoalberi. L'uso delle parentesi quadre e tonde non ha alcun significato semantico, ma serve soltanto a rendere la rappresentazione più facile da interpretare. Senza l'indicazione dei sottoalberi la scrittura diventa:

$$v_1 \quad v_2 \quad v_5 \quad v_6 \quad v_{10} \quad v_3 \quad v_4 \quad v_7 \quad v_{11} \quad v_{12} \quad v_8 \quad v_9$$

17.10.3 Visita in ordine differito

La visita in *ordine differito* prevede le seguenti azioni:

1. se $n \geq 0$ è il grado (numero di sottoalberi) della radice, allora
   - visita il primo sottoalbero, in ordine differito;
   - visita il secondo sottoalbero, in ordine differito;
   - ...
   - visita l'n-esimo sottoalbero, in ordine differito;
2. esamina la radice

Si noti che in questo caso la radice viene esaminata *dopo* aver esaminato i relativi sottoalberi.

In Figura 17.9 è riportata la sequenza determinata dalla visita differita dell'albero di Figura 17.7. La scrittura alternativa della sequenza di visita è il seguente:

$$[[((v_5) \quad (v_{10} \quad (v_6))) \quad v_2] \quad [v_3] \quad [(((v_{11}) \quad (v_{12})) \quad v_7) \quad (v_8) \quad (v_9)] \quad v_4] \quad v_1$$

Anche in questo caso, le parentesi sono utilizzate per evidenziare i sottoalberi. Senza l'indicazione dei sottoalberi la scrittura diviene:

$$v_5 \quad v_{10} \quad v_6 \quad v_2 \quad v_3 \quad v_{11} \quad v_{12} \quad v_7 \quad v_8 \quad v_9 \quad v_4 \quad v_1$$

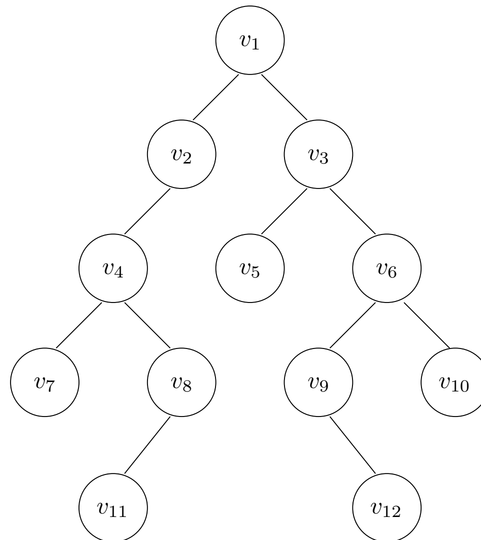


Figura 17.10 Esempio di albero binario.

17.11 ALBERI BINARI

Un *albero binario* è un insieme di nodi, tale che:

- un particolare nodo, se il numero dei nodi è diverso da zero, è designato come radice
- i rimanenti nodi, se esistono, possono essere suddivisi in *due* insiemi disgiunti, ciascuno dei quali è a sua volta un albero binario (sottoalbero sinistro e destro)

Si noti che l'albero binario *non* è un caso particolare di albero, per i seguenti due motivi:

- un albero binario può essere vuoto, mentre un albero deve contenere almeno un nodo
- ciascuno dei due sottoalberi della radice conserva la propria identità di sottoalbero destro e sinistro anche se l'altro sottoalbero è vuoto

La Figura 17.10 riporta un esempio di albero binario. Questo vincolo è molto più restrittivo dell'ordinamento dei nodi nei livelli di un albero ordinato. Ad esempio, due alberi contenenti un solo nodo, oltre alla radice, sono distinti se nel primo tale nodo è designato come sottoalbero sinistro della radice e nel secondo come sottoalbero destro. I metodi di visita anticipato e differito si applicano agli alberi binari ai quali si applica anche la visita in ordine simmetrico.

17.11.1 Visita in ordine simmetrico

La visita in *ordine simmetrico* prevede le seguenti azioni:

- visita il sottoalbero sinistro, in ordine simmetrico;
- esamina la radice;
- visita il sottoalbero destro, in ordine simmetrico.

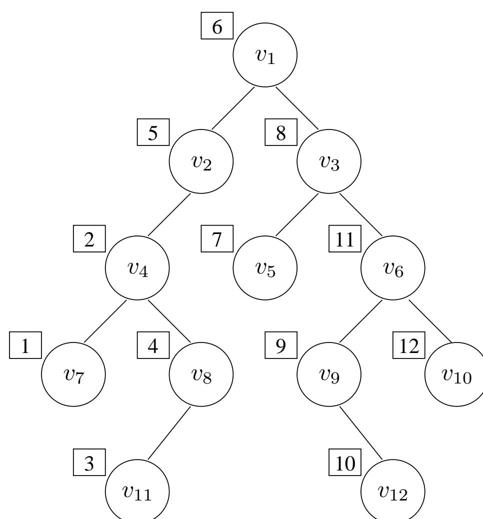


Figura 17.11 Visita in ordine simmetrico dell'albero binario di Figura 17.10.

In Figura 17.11 è riportato un esempio di visita in ordine simmetrico di un albero binario. La scrittura equivalente alla sequenza di visita è la seguente:

$$v_7 \quad v_4 \quad v_{11} \quad v_8 \quad v_2 \quad v_1 \quad v_5 \quad v_3 \quad v_9 \quad v_{12} \quad v_6 \quad v_{10}$$

Dal momento che l'albero binario è un caso particolare di albero, è dunque possibile visitarlo in ordine anticipato e differito. Per l'albero di Figura 17.11 tali sequenze di visita sono

$$v_1 \quad v_2 \quad v_4 \quad v_7 \quad v_8 \quad v_{11} \quad v_3 \quad v_5 \quad v_6 \quad v_9 \quad v_{12} \quad v_{10}$$

e

$$v_7 \quad v_{11} \quad v_8 \quad v_4 \quad v_2 \quad v_5 \quad v_{12} \quad v_9 \quad v_{10} \quad v_6 \quad v_3 \quad v_1$$

rispettivamente per l'ordine anticipato e differito.

L'importanza dell'albero binario è dovuta alla relativa semplicità con cui tale struttura può essere allocata in memoria e alla possibilità di rappresentare qualsiasi albero ordinato in forma di albero binario. Per ottenere questa trasformazione esiste la regola seguente:

Un albero ordinato S si rappresenta come albero binario T, se:

- gli insiemi di nodi di T e S coincidono;
- la radice di T coincide con la radice di S;
- ogni nodo dell'albero binario T:
  - ha come figlio sinistro il primo figlio del nodo omonimo dell'albero S;
  - ha come figlio destro il fratello del nodo omonimo dell'albero S

Capitolo 18

STRUTTURE CONCRETE DI DATI

IL problema è quello di rappresentare nella memoria del calcolatore, formata, come è noto, da celle di memoria a ciascuna delle quali è associato un indirizzo, le strutture astratte di dati presentate nel Capitolo 17. L'organizzazione della memoria è estremamente elementare e mal si presta alla memorizzazione delle informazioni delle strutture astratte: la soluzione è quella di realizzare dei programmi che permettano all'utente di impiegare la macchina come se tali strutture fossero proprie della macchina

Le strutture concrete comprendono:

- la struttura sequenziale
- la catena o lista
- il plesso

18.1 STRUTTURA SEQUENZIALE

La *struttura sequenziale* è la struttura concreta più semplice ed intuitiva. È dotata delle seguenti caratteristiche:

- è un insieme di elementi omogenei (cioè del medesimo tipo e quindi dimensione) disposti in modo adiacente in memoria, collocati ad indirizzi crescenti; ciascun elemento può richiedere una o più celle;
- tipicamente tutti gli elementi hanno la medesima lunghezza, cioè occupano lo stesso numero di celle.

Figura 18.1 Esempio di memorizzazione di una struttura sequenziale.

I parametri che caratterizzano una struttura sequenziale o vettore sono:

- l'indirizzo $addr_b$ del primo elemento, detto indirizzo base del vettore;
- il numero m dei suoi elementi, cioè la lunghezza del vettore;
- il numero d di celle occupate da ciascun elemento, ovvero la dimensione dell'elemento

Di conseguenza l'indirizzo del primo elemento della struttura sequenziale è:

$$addr(x_0) = addr_b$$

Mentre l'indirizzo dell'elemento x_i si otterrà dal calcolo:

$$addr(x_i) = addr(x_0) + i * d$$

La lunghezza m deve essere fissata e non può essere modificata. La struttura sequenziale è quindi una struttura rigida, adatta a contenere serie di dati il cui numero sia noto a priori, o per cui si possa prevedere un limite superiore.

In Figura 18.1 è rappresentato un esempio di memorizzazione di una struttura sequenziale composta da 12 elementi di dimensione pari a 6 blocchi elementari ciascuna (gli indici variano da 0 a 11). Si noti che le dimensioni vengono espresse in "blocchi elementari" e non in byte, in quanto la dimensione del blocco elementare dipende dall'architettura del calcolatore in uso. Tipicamente si hanno valori di 8, 16, 32 e 64 bit per blocco elementare a seconda della macchina sulla quale si deve memorizzare l'informazione. Nell'esempio, l'indirizzo del primo elemento è $addr(x_0) = 500$, mentre l'indirizzo dell'i-esimo elemento è $addr(x_i) = 500 + 6 * i$. Ad esempio, l'indirizzo di partenza dell'elemento x_8 è pari a $500 + 6 * 8 = 548$.

Nel caso in cui gli elementi da memorizzare in una struttura sequenziale abbiano diversa lunghezza, si possono adottare due soluzioni:

1. normalizzare le lunghezze degli elementi riportandole alla lunghezza dell'elemento più lungo;
2. dotare ogni elemento delle informazioni sufficienti a determinare l'indirizzo dell'elemento successivo, ad esempio un numero che rappresenta la lunghezza dell'elemento.

	0	1	2	3	4	5	6	7	8	9
500	5		elemento 0				4		elemento 1	
510		2	elemento 2		3	elemento 3			3	e
520	lemento 4		4		elemento 5			6		
530	elemento 6				4		elemento 7			2
540	elemento 8		5			elemento 9			2	eleme
550	nto 10	3		elemento 11						
560										
570										
580										

Figura 18.2 Esempio di memorizzazione di una struttura sequenziale con elementi di dimensione variabile.

Si supponga di voler memorizzare 12 elementi di dimensione variabile tra 2 e 6 blocchi elementari. Se si adotta la prima soluzione, una possibile allocazione della memoria è la stessa che nell'esempio di Figura 18.1. Ciascun blocco è infatti stato ridimensionato sulla base dell'elemento più lungo, che è proprio di 6 blocchi elementari. La stessa struttura è stata memorizzata con la seconda tecnica, e una sua possibile allocazione è stata rappresentata in Figura 18.2. A ciascun blocco di dati è stato aggiunto un blocco iniziale nel quale viene memorizzata la lunghezza totale dell'elemento, *esclusa l'intestazione*. Come si può notare, anche se nel caso pessimo un elemento può raggiungere la dimensione di 7 blocchi, l'occupazione totale di memoria è inferiore al caso precedente. La scelta di una tecnica piuttosto che l'altra dipenderà quindi da considerazioni sulla lunghezza media dei blocchi: quanto più la media si avvicinerà alla lunghezza massima, tanto più sarà conveniente l'uso della prima tecnica.

È possibile valutare la convenienza dell'uso di una o dell'altra tecnica con semplici calcoli. Per esempio, dovendo memorizzare m elementi di dimensione massima pari a d, la prima tecnica prevede di usare m elementi tutti di dimensione pari a quella massima, per un totale di

$$mem_1 = d \cdot m$$

blocchi occupati. Per contro, la seconda tecnica aggiunge un blocco ad ogni elemento, quindi la sua occupazione di memoria risulta essere pari a

$$mem_2 = (\overline{d} + 1)m$$

dove $\overline{d}$ indica la dimensione media dei blocchi, ovvero

$$\overline{d} = \frac{\sum_{i=0}^{m-1} d_i}{m} \quad m > 0$$

dove d_i indica la dimensione dell'i-esimo elemento.

Lo svantaggio principale delle strutture sequenziali è la scarsa flessibilità. Infatti:

- l'inserimento di un nuovo elemento tra due elementi preesistenti richiederebbe lo spostamento in avanti di tutti gli elementi che devono seguire l'elemento inserito; avanti

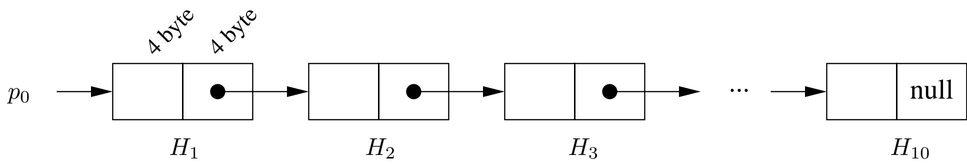


Figura 18.3 Esempio di lista.

	0	1	2	3	4	5	6	7	8	9
500	val(H5)	537	val(H2)	568						
510									val(H8)	521
520		val(H9)	581							
530								val(H6)	575	
540					val(H1)	502				
550			val(H4)	500						
560									val(H3)	552
570						val(H7)	518			
580		val(H10)	null							

Figura 18.4 Esempio di allocazione in memoria della lista.

- l'eliminazione di un elemento richiede lo spostamento di tutti gli elementi che seguono tale elemento per "compattare" la struttura, se non si vogliono lasciare dei "buchi" inutilizzati

Il vantaggio delle strutture sequenziali è dato dalla semplicità di gestione e dall'efficienza di memorizzazione, in quanto non sono richieste informazioni supplementari per gestirle.

18.2 CATENA O LISTA

La *catena* o *lista* permette di assegnare alle celle un ordinamento logico arbitrario, differente dal loro ordinamento fisico, indicato dagli indirizzi. Si tratta di un insieme di elementi disposti in modo arbitrario nella memoria, purché non sovrapposti, nel quale ogni elemento è costituito da due parti:

- il *dato* che rappresenta l'elemento della struttura astratta da rappresentare
- l'*indirizzo* dell'elemento successivo della catena (puntatore)

L'ultimo elemento contiene un puntatore nullo per indicare che non ci sono altri elementi nella lista.

Un esempio di lista è riportata in Figura 18.3. Essa è costituita da 10 elementi, ciascuno delle lunghezza di 8 byte: 4 byte contengono l'informazione memorizzata, mentre gli altri 4 contengono il puntatore all'elemento successivo. In Figura 18.4 è riportato un esempio di come i singoli elementi sono possono essere allocati in memoria, con il relativo valore dei puntatori agli elementi successivi. Il

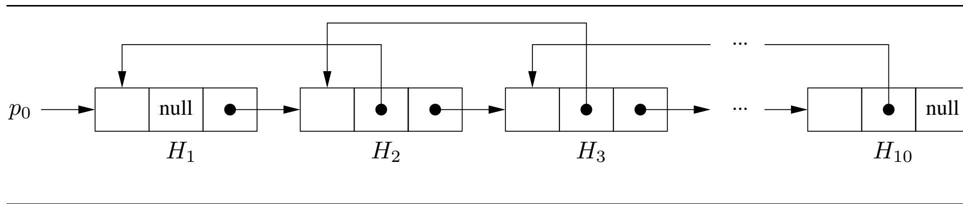


Figura 18.5 Esempio di lista bidirezionale.

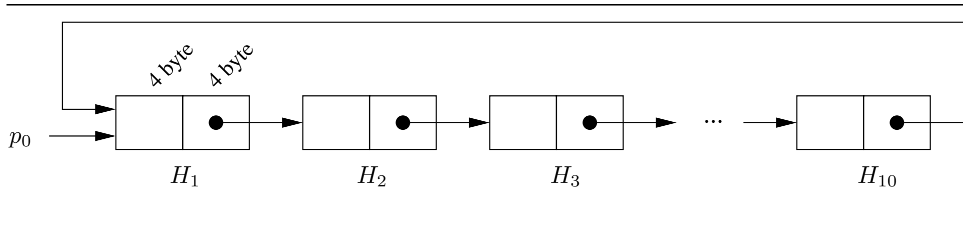


Figura 18.6 Esempio di lista circolare.

puntatore associato all'elemento H_{10} è nullo, in quanto H_{10} è l'ultimo della lista. A costo di risultare ovvii, è da notare che è perfettamente lecito avere un elemento posizionato ad un indirizzo che finisce con la cifra 9 (ultima colonna della rappresentazione); in tal caso l'elemento corrispondente sarebbe da rappresentare per metà alla fine della riga e per metà all'inizio della riga successiva. Questo per sottolineare che, anche se nella figura la memoria è rappresentata in forma di matrice, in realtà è da considerarsi come una successione contigua di celle, anche se per esigenze di rappresentazione ciò non è stato fatto.

Il reperimento delle informazioni avviene attraverso una scansione della catena: l'indirizzo di un elemento è noto sotto forma di puntatore contenuto nell'elemento precedente.

È possibile realizzare catene o *liste bidirezionali*, le quali sono costituite da elementi dotati anche di puntatori all'elemento precedente. Un esempio di lista bidirezionale è rappresentato in Figura 18.5.

La catena o *lista circolare*, o ciclica, ha nell'ultimo elemento un puntatore al primo elemento. Un esempio di lista bidirezionale è rappresentato in Figura 18.6.

18.2.1 Inserimento ed eliminazione di elementi

L'inserimento di un elemento H_{new} tra gli elementi H_i e H_{i+1} richiede le operazioni:

$$ptr(H_{new}) \leftarrow addr(H_{i+1})$$

$$ptr(H_i) \leftarrow addr(H_{new})$$

Si noti che la prima operazione equivale a:

$$ptr(H_{new}) \leftarrow ptr(H_i)$$

L'operazione di inserimento è schematizzata in Figura 18.7, nella quale la freccia tratteggiata rappresenta il puntatore precedente.

L'eliminazione di un elemento H_i richiede la scansione fino all'elemento H_{i-1} e quindi l'operazione:

$$ptr(Hi - 1) \leftarrow addr(H_{i+1})$$

che equivale a

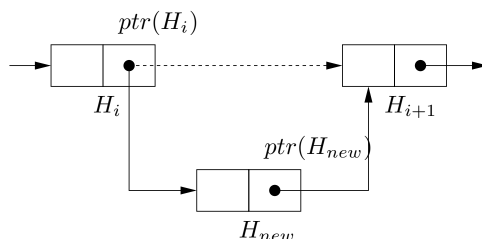


Figura 18.7 Inserimento dell'elemento H_{new} in una lista.

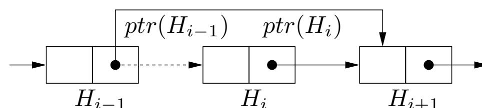


Figura 18.8 Eliminazione dell'elemento H_i dalla lista.

$$ptr(H_{i-1}) \leftarrow addr(H_i)$$

L'operazione di eliminazione è schematizzata in Figura 18.8, nella quale la freccia tratteggiata rappresenta il puntatore precedente. Quando un elemento viene eliminato dalla lista c'è il problema che esso è comunque sempre presente in memoria, ma non è più utilizzato nè raggiungibile.

È necessario disporre di un metodo di amministrazione della memoria libera, responsabile della raccolta delle celle che si rendono libere in seguito all'eliminazione di elementi della lista. Esistono diverse possibilità: una prevede di formare una catena con le celle libere, detta catena libera, e di gestirla con le regole delle catene. Da questa lista si estraggono gli elementi necessari a memorizzare nuove informazioni, e in essa si re-inseriscono nuovi elementi non più necessari, seguendo i procedimenti di eliminazione ed inserimento. I vantaggi di questa soluzione sono:

- compatta memorizzazione di insiemi di dati per cui è richiesto un ordinamento diverso da quello in cui i dati si presentano
- inserimento ed eliminazione di elementi molto semplice
- occupazione di memoria buona se si usano algoritmi di gestione della memoria libera

Gli svantaggi invece sono:

- spreco di spazio derivante dai puntatori, tanto più rilevante quanto più sia piccolo il campo contenente l'informazione
- necessità di accompagnare ogni operazione con il relativo aggiornamento della lista libera
- inefficienza nell'accesso ad un elemento, che richiede la scansione della lista

Le considerazioni relative alla diversa dimensione degli elementi di una struttura sequenziale si possono estendere anche alle catene: anche in questo caso si può aggiungere ad ogni elemento un ulteriore campo in cui sia specificata la dimensione dell'elemento stesso.

La catena o *lista multipla* è una generalizzazione del caso semplice in cui viene aggiunto un campo che contiene un puntatore verso una lista che a sua volta contiene l'informazione.

18.3 MEMORIZZAZIONE DELLE STRUTTURE ASTRATTE: LISTE

Le liste possono essere rappresentate facilmente sia mediante strutture sequenziali, sia mediante strutture concatenate. La scelta di uno e dell'altro tipo di struttura concreta dipende dalle operazioni che devono essere fatte sulle liste.

Se le operazioni consistono nella lettura degli elementi della lista, con eventuale modifica, la memorizzazione sequenziale è idonea. Diverso è il discorso quando si eseguono operazioni che alterano l'ordine degli elementi (inserimenti, eliminazioni, fusioni di liste); in tal caso, l'uso di strutture concatenate è più adatto.

18.4 MEMORIZZAZIONE DELLE STRUTTURE ASTRATTE: CODE, PILE, DOPPIE CODE

Possono essere usate sia la rappresentazione sequenziale, sia quella concatenata; sono comunque necessari uno o più puntatori, modificati dopo ogni inserimento o eliminazione, che identificano il primo, l'ultimo elemento, o entrambi, all'interno della struttura.

In particolare la coda è in genere rappresentata con una catena circolare con un puntatore che tiene aggiornato l'indirizzo del fondo della coda: in tale elemento viene tenuto l'indirizzo della testa della coda. La memorizzazione di una pila, invece, può essere fatta con una struttura sequenziale. Infatti la testa della pila rimane fissa ed è quindi sufficiente un puntatore sull'ultimo elemento (elemento affiorante della pila) da aggiornare dopo ogni inserimento o eliminazione. Per la doppia coda, la soluzione più conveniente è l'uso di una catena bidirezionale con due puntatori.

18.5 MEMORIZZAZIONE DELLE STRUTTURE ASTRATTE: MATRICI

Si usa tipicamente una struttura sequenziale accodando gli elementi riga dopo riga oppure colonna dopo colonna. Per una matrice avente m righe e n colonne, l'indirizzo del generico elemento A_{ij} è dato da

$$addr(A_{ij}) = addr(A_{11}) + (i-1)*n*l + (j-1)*l$$

dove:

- $addr(A_{11})$ è l'indirizzo iniziale della struttura sequenziale
- l è la lunghezza di ciascun elemento

Se la memorizzazione avviene per colonne, basta sostituire nell'equazione m ad n e i a j. Il ragionamento si può estendere anche a matrici aventi k dimensioni. Se la matrice ha grandi dimensioni, ma ha molti zeri (matrice sparsa), può essere opportuno organizzare gli elementi non nulli, insieme con i loro indici, in una *tavola*.

Un'altra possibilità è quella di usare una doppia famiglia di catene circolari: ogni elemento non nullo appartiene a 2 catene, una di riga ed una di colonna; quindi l'elemento della catena è formato:

- dal dato
- dai suoi due indici
- da due puntatori agli elementi successivi su riga e colonna

18.6 MEMORIZZAZIONE DELLE STRUTTURE ASTRATTE: TAVOLE

Sono memorizzate tipicamente in strutture sequenziali. I metodi per l'accesso ai singoli elementi possono essere molto diversi in funzione dell'uso previsto per le tavole stesse: la scelta del metodo ha l'obiettivo di minimizzare la lunghezza di ricerca, cioè il numero di chiavi esaminate prima di raggiungere l'elemento corrispondente ad una chiave prefissata. I metodi più diffusi sono:

- ricerca sequenziale
- ricerca binaria
- accesso diretto
- accesso calcolato (hash)

Questi metodi di ricerca verranno brevemente accennati nelle seguenti sezioni, e verranno ripresi più approfonditamente nel Capitolo 19.

18.7 RICERCA SEQUENZIALE

Nella ricerca sequenziale l'operazione di ricerca si effettua scandendo gli elementi della tavola fino al reperimento della chiave desiderata.

Sebbene sia banale, questo tipo di ricerca ha il vantaggio di permettere una memorizzazione degli elementi nella tavola senza seguire alcuna regola di ordinamento. Si tratta però di una tecnica scarsamente efficiente, in cui la complessità media di ricerca è pari a N/2, se N è il numero di elementi memorizzati nel vettore. La complessità media di ricerca può essere calcolata come la media del numero di accessi al vettore richiesti per trovare un elemento che si trovi in un punto qualsiasi della tavola. Diverse tecniche possono essere applicate per migliorare la velocità di accesso, per esempio concentrando gli elementi in cui si prevede un accesso frequente nelle prime posizioni accedute.

Una tavola su cui si opera una ricerca sequenziale può essere memorizzata come struttura sequenziale o a catena.

18.8 RICERCA BINARIA

Per poter effettuare una ricerca binaria la tavola deve essere ordinata secondo le chiavi. La ricerca binaria richiede il confronto fra la chiave cercata e quella dell'elemento centrale della tavola. Il risultato del confronto indica se la ricerca ha termine oppure in quale metà della tavola deve continuare. Il procedimento viene iterato fino all'individuazione della chiave oppure fino a quando l'intervallo di ricerca si anulla.

Il tempo massimo di ricerca è $log_2 N$. È una tecnica di ricerca veloce. Se la tavola è memorizzata in una struttura sequenziale, può essere difficile inserire nuovi elementi, in quanto è richiesto il riordino delle chiavi per mantenerne appunto l'ordinamento.

Una tavola su cui si opera una ricerca binaria che venga memorizzata in una struttura sequenziale richiede che gli elementi abbiano lunghezza costante, dato che l'accesso deve avvenire tramite calcolo dell'indirizzo.

18.9 ACCESSO DIRETTO

L'accesso diretto è applicabile solo a tavole per cui le chiavi permettano di stabilire una corrispondenza biunivoca con gli indirizzi di memoria corrispondenti agli elementi della tavola.

Esiste quindi una funzione di accesso che, a partire dalla chiave dell'elemento ricercato, permette di ottenere l'indirizzo di memoria. A chiavi diverse corrispondono indirizzi diversi. È però difficile prevedere l'inserimento di nuovi elementi, in quanto le relative chiavi devono ancora permettere l'identificazione di indirizzi diversi.

Questo metodo è caratterizzato da una altissima velocità di accesso, ma scarsa applicabilità a causa delle ipotesi restrittive per il calcolo dell'indirizzo dei singoli elementi.

18.10 ACCESSO CALCOLATO (HASHING)

È una generalizzazione dell'accesso diretto, essendo basato sulla funzione di accesso e richiedendo che la tavola sia memorizzata in una struttura sequenziale; tuttavia consente l'agevole inserimento di nuovi elementi.

Il metodo si basa su una funzione di accesso, che applicata alla chiave fornisce un numero intero minore della lunghezza della struttura sequenziale, che è a sua volta maggiore del numero di elementi da memorizzare. Il numero rappresenta l'indice dell'elemento all'interno della struttura.

Purtroppo, a coppie di chiavi distinte può corrispondere lo stesso numero, cioè si ha il cosiddetto problema della collisione di elementi. Le chiavi che generano delle collisioni sono dette *sinonimi*. Nasce quindi il problema della scelta delle posizioni nella struttura sequenziale corrispondenti agli elementi in collisione. Per esempio, i sinonimi possono essere memorizzati in catene interne o esterne alla tavola. Senza entrare nel dettaglio delle possibili tecniche di gestione dei sinonimi, è da tenere ben presente che la scelta della funzione di accesso ed il criterio di gestione delle collisioni determinano la bontà dell'implementazione della tavola.

18.11 MEMORIZZAZIONE DI ALBERI E GRAFI IN CATENE

Possono essere rappresentati da una catena generalizzata, in cui nella catena principale compaiono i dati associati a tutti i nodi, accompagnati da due puntatori:

1. il primo punta ad una catena secondaria in cui compaiono tanti elementi quanti sono i nodi adiacenti a quello in esame
2. il secondo punta all'elemento successivo nella catena principale

La Figura 18.10 visualizza la memorizzazione del grafo riportato in Figura 18.9. Ciascun nodo contiene il dato da memorizzare e una lista di puntatori ai nodi ai quali è collegato da un arco.

18.12 MEMORIZZAZIONE DI ALBERI E GRAFI IN PLESSI

Si possono usare anche i plessi: ogni elemento contiene il dato del nodo e tanti puntatori quanti sono i nodi adiacenti a quello in esame.

Il plesso è particolarmente adatto a rappresentare alberi binari: in tal caso, si hanno sempre due nodi uscenti da ogni nodo dell'albero e, di conseguenza, il formato degli elementi del plesso è omogeneo e può essere riportato una volta per tutte al di fuori degli elementi del plesso.

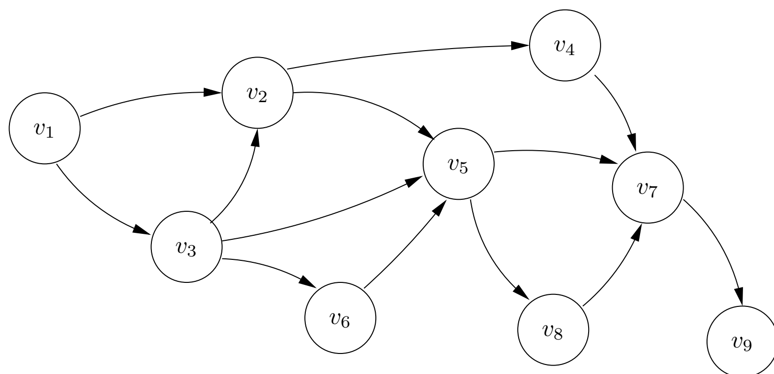


Figura 18.9 Grafo di esempio.

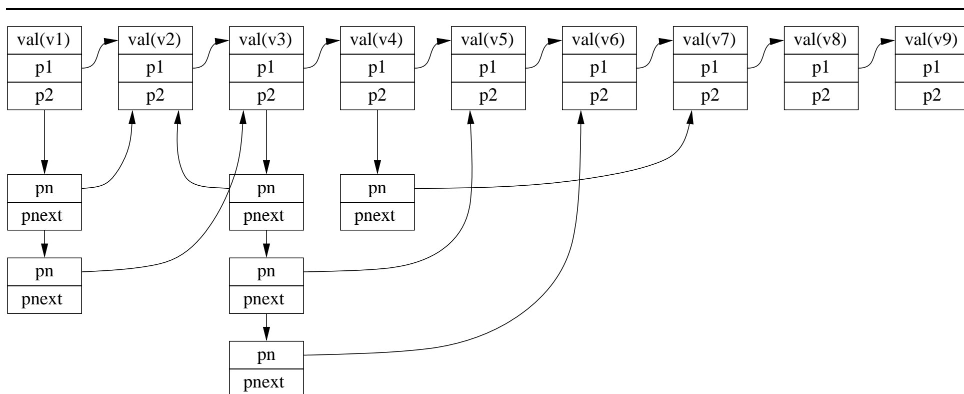


Figura 18.10 Esempio di memorizzazione del grafo di Figura 18.9 (sono visualizzati soltanto una parte dei collegamenti).

Capitolo 19

TECNICHE DI PROGRAMMAZIONE E ALGORITMI BASE

In questo capitolo sono presentate alcuni algoritmi di base per lo sviluppo di algoritmi più complessi. Saranno presentati in particolare gli algoritmi di ricerca e di ordinamento.

19.1 L'ORDINAMENTO

Un algoritmo di ordinamento ha lo scopo di ordinare un insieme di valori. Per il funzionamento di un algoritmo di ordinamento, deve essere possibile:

- mantenere gli elementi in un opportuno vettore che ne conservi l'ordine
- stabilire una relazione d'ordine tra due elementi dell'insieme da ordinare
- scambiare due elementi qualsiasi

Tipicamente si tratta di ordinare valori numerici, ma questi possono essere visti come chiavi di strutture dati complesse, che ne permettono quindi l'ordinamento.

19.1.1 Bubblesort

Uno dei più semplici esempi di algoritmo di ordinamento è il *bubble sort*, che si può implementare come nell'esempio seguente[31]:

[31] Questa implementazione è una versione leggermente ottimizzata della versione orginale dell'algoritmo.

Il programma è contenuto nel file `bubble_sort.c`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N   (20)
5  #define MAX (20)
6
7  void swap(int *a, int *b)
8  {
9    int tmp;
10
11   tmp = *a;
12   *a = *b;
13   *b = tmp;
14 }
15
16 void bubblesort(int l[], int n)
17 {
18   int i, j;
19
20   for (i = 0; i < n; i++) {
21     for (j = i + 1; j < n; j++) {
22       if (l[i] > l[j]) {
23         swap(&l[i], &l[j]);
24       }
25     }
26   }
27 }
28
29 int main(int argc, char **argv)
30 {
31   int i;
32   int l[N];
33
34   printf("Non ordinati: \n");
35   for (i = 0; i < N; i++) {
36     l[i] = rand() % MAX;
37     printf("%d ", l[i]);
38   }
39   printf("\n");
40
41   bubblesort(l, N);
42
43   printf("Ordinati: \n");
44   for (i = 0; i < N; i++) {
45     printf("%d ", l[i]);
46   }
47   printf("\n");
```

```
48
49     return 0;
50   }
```

La funzione di ordinamento `bubblesort` accetta in ingresso un vettore di valori interi da ordinare, passato nella forma di puntatore, e il numero di elementi che costituiscono il vettore. Vengono in questo caso scanditi tutti gli elementi del vettore e vengono confrontati con *tutti i seguenti*, e in caso il test indichi che l'ordinamento non è quello desiderato, i valori vengono scambiati. Così facendo, i valori vengono man mano spostati rispettando l'ordinamento reciproco.

Nel programma di esempio viene riempito un vettore di elementi generati casualmente, mediante la funzione `rand` e l'operatore modulo. Il vettore viene ordinato con la chiamata a `bubblesort`, e il risultato viene visualizzato a video.

19.2 ALGORITMI DI RICERCA

Un algoritmo di ricerca permette di trovare un elemento all'interno di un elenco, e di determinare se eventualmente tale elemento non esiste all'interno dell'elenco.

In particolare, se la ricerca avviene su un vettore di elementi si sarà interessati all'indice dell'elemento desiderato, mentre se la ricerca avviene su una lista, il risultato della ricerca sarà costituito dall'indirizzo in memoria dell'elemento, ovviamente qualora la ricerca abbia successo.

19.2.1 La ricerca sequenziale

La *ricerca sequenziale* è il metodo di ricerca più intutitivo e di immediata realizzazione. Esso permette di effettuare una ricerca su un qualsiasi insieme di dati, semplicemente scandendo tutti gli elementi e confrontandoli con l'elemento desiderato fino a trovare una corrispondenza. Se tutti gli elementi vengono confrontati senza trovare una corrispondenza, l'algoritmo termina con un insuccesso.

Nella seguente implementazione, il vettore `l` viene scandito fino a trovare un elemento che corrisponda all'elemento `x` desiderato, oppure finché tutti gli elementi non sono stati esaminati (`i < n`). All'uscita del ciclo `while` si controlla il valore dell'indice `i`: se questo è inferiore ad `n`, allora l'elemento è stato trovato, altrimenti no.

Le funzioni di ricerca sequenziale e binaria sono contenute nel file di esempio `ricerca.c`.

```
int ssearch(int l[], int x, int n)
{
  int i = 0;

  while ((i < n) && (x != l[i])) i++;

  if (i < n)
    return i;

  return -1;
}
```

Una tecnica simile per realizzare la ricerca sequenziale è possibile impostando un valore "sentinella", che viene utilizzato per controllare e determinare la fine del ciclo. Invece di effettuare il test sul numero di elementi esaminati, si controlla l'uguaglianza col valore sentinella, e nel caso l'uguaglianza sia verificata, il, ciclo termina. Un esempio di tale algoritmo è il seguente:

```
int ssearch2(int l[], int x, int n)
{
  int i;

  l[n] = x;

  for (i = 0; l[i] != x; i++);

  if (i < n)
    return i;

  return -1;
}
```

Gli algoritmi di ricerca sequenziale hanno complessità pari a n nel caso peggiore, e mediamente pari a $n/2$, dove n è il numero di elementi sul quale effettuare la ricerca.

19.2.2 La ricerca binaria

Quando è possibile sfruttare qualche caratteristica favorevole dell'insieme di elementi sul quale effettuare la ricerca, è possibile realizzare degli algortimi di ricerca più efficienti della ricerca sequenziale. La *ricerca binaria* si può applicare ad un *insieme ordinato* di elementi, sfruttando appunto la proprietà di ordinamento dell'insieme.

L'idea di base è quella di dividere l'intervallo di ricerca in due metà. Viene confrontato l'elemento di mezzo `m` con l'elemento desiderato `x`: se coincidono allora la ricerca ha avuto successo, altrimenti si confronta `x` con `m` e si procede alla ricerca nella metà superiore o inferiore dell'intervallo a seconda che `x` sia rispettivamente maggiore o minore di `m`. La ricerca fallisce quando l'intervallo di ricerca di annulla e l'elemento desiderato non è stato ancora trovato.

Di seguito è illustrato l'algoritmo di ricerca binaria in forma ricorsiva che serve per trovare un intero `x` all'interno di un vettore ordinato `l` costituito da `n` elementi[32].

```
int bsearch1(int l[], int x, int a, int b)
{
  int m;

  /* elemento centrale */
  m = (a + b) / 2;

  if ((m < a) || (b < 0)) {
    return -1;                     /* l'elemento desiderato non c'e' */
  } else if (x < l[m]) {
    return bsearch1(l, x, a, m - 1);  /* ricerca nella parte inferiore */
  } else if (x > l[m]) {
    return bsearch1(l, x, m + 1, b);  /* ricerca nella parte superiore */
  } else {
    return m;                      /* indice dell'elemento desiderato */
  }
}
```

[32]La funzione `bsearch1` è così denominata, invece che nel modo più naturale di `bsearch`, poiché nella libreria standard è presente una versione più generica dell'algoritmo di ricerca, utilizzabile includendo il file di intestazione `stdlib.h`.

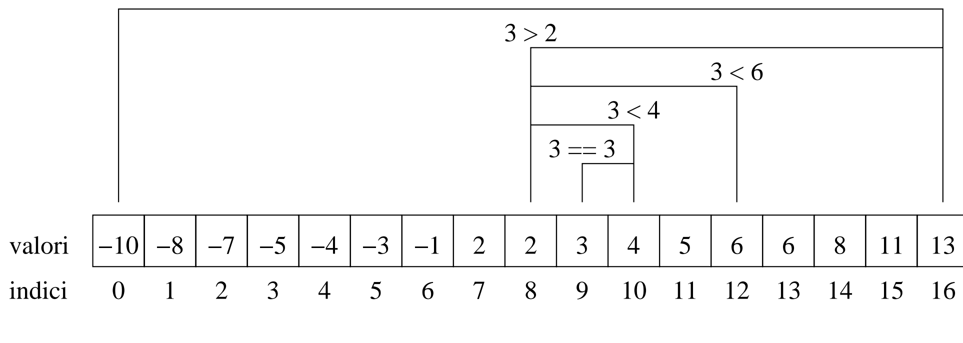


Figura 19.1 Esempio di ricerca binaria.

Si faccia attenzione al fatto che, nell'implementazione, i valori `a` e `b` che delimitano l'intervallo di ricerca sono degli *indici* per il vettore `l`, e non i valori contenuti in determinate posizioni del vettore.

In Figura 19.1 è rappresentato un esempio di ricerca binaria su un vettore di 17 elementi, nel quale si è interessati a trovare l'elemento di valore 3.

Il vantaggio della ricerca binaria rispetto a quella sequenziale è che richiede al più $\log_2(n) + 1$ iterazioni per determinare la posizione dell'elemento desiderato o per stabilire che esso non sia presente.

Dal momento che qualsiasi algoritmo ricorsivo può essere implementato in forma non ricorsiva, di seguito è presentata la forma non ricorsiva dell'algoritmo di ricerca binaria. Ad ogni passo viene ricalcolato esplicitamente il limite superiore o inferiore dell'intervallo di ricerca, e l'elemento centrale viene confrontato con il valore desiderato.

Quando l'esecuzione esce dal ciclo `while` significa che l'elemento `x` non è presente in lista.

```
int bsearch1_nr(int l[], int x, int n)
{
  int a, b, m;

  a = 0;
  b = n - 1;

  while (a <= b) {
    m = (a + b) / 2;
    if(l[m] == x)
      return m;   /* valore x trovato alla posizione m */
    if(l[m] < x)
      a = m + 1;
    else
      b = m - 1;
  }

  return -1;
}
```

Lo svantaggio è che gli elementi del vettore devono essere ordinati, e questo comporta un onere di calcolo "a monte" dell'algoritmo di ricerca.

19.2.3 Hashing

La ricerca basata sull'hashing richiede che una funzione, la cosiddetta *funzione di hash*, a partire dal valore o della chiave che deve essere cercata, calcoli un indirizzo o indice al quale trovare l'elemento desiderato. Nel caso semplificato di ricerca di una stringa in un vettore di stringhe, la funzione dovrà restituire l'indice all'interno del vettore alla quale si trova (se mai vi è stata memorizzata) la stringa desiderata.

Perché la ricerca avvenga correttamente, la funzione di hash deve anche essere utilizzata per memorizzare i dati: quando si desidera memorizzare un valore, si calcola sempre mediante la funzione di hash, l'indice al quale memorizzare il dato. Così facendo, dato un determinato valore da cercare/memorizzare, si opererà la ricerca sempre allo stesso indice nel vettore.

Il problema dell'hashing è dovuto al fatto che più chiavi possono dare luogo allo stesso indice, cioè si hanno le cosiddette collisioni. Ovviamente soltanto un valore può essere memorizzato in una data posizione di un vettore, quindi bisogna gestire opportunamente il caso in cui due o più valori generino lo stesso indice.

Le soluzioni sono varie, più o meno efficienti, eleganti ed efficaci. Per esempio, per ogni posizione del vettore, invece di memorizzare direttamente un elemento, è possibile mantenere una lista di elementi; il valore corrispondente all'indice desiderato sarà memorizzato nella lista insieme a tutti gli altri elementi che generano collisioni per quell'indice. La ricerca sulla lista può essere fatta con uno qualsiasi dei metodi illustrati. Un altro metodo consiste, in caso di collisione nell'inserimento di un valore, nel memorizzare ad indici successivi a quello generato dalla funzione di hash. Infine, in alcuni casi, può semplicemente essere sufficiente non gestire affatto le collisioni, e memorizzare nella posizione generata dalla funzione di hash soltanto il primo o l'ultimo dei valori che hanno prodotto una collisione, perdendo le informazioni riguardo agli altri elementi.

Il programma illustrato come esempio è molto semplice, e si è fatta la scelta di non gestire le collisioni. In caso di collisione, viene memorizzato soltanto l'ultimo dei dati inseriti, perdendo così le informazioni riguardanti dati precedentemente memorizzati.

Il programma è contenuto nel file `hash.c`.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #define N1 (8)
5  #define N2 (10)
6
7  char *str[N1] = {
8    "Roma",
9    "Milano",
10   "Palermo",
11   "Torino",
12   "Genova",
13   "Napoli",
14   "Firenze",
15   "Ramo",
16 };
17
18 char *v[N2];
19
20 int hash(char *s, int n);
```

```
21 int trova(char *s);
22 void memorizza();
```

Il programma utilizza per semplicità soltanto variabili globali. È presente un vettore `str` di `N1` stringhe, le quali sono le stringhe da memorizzare nel vettore di hashing `v`, a sua volta in grado di contenere fino a `N2` stringhe. In realtà il vettore `v` non memorizzerà la stringha, ma manterrà un puntatore alla stringa contenuta in `str`. Il vettore `v`, essendo una variabile globale, ha tutti gli elementi inizializzati automaticamente a zero, cioè, essendo puntatori, a `NULL`.

Vengono poi dichiarate le tre funzioni utilizzate nel programma: la funzione di hash, la funzione `torva` per la ricerca di un elemento nel vettore, e la funzione `memorizza` per inserire tutte le stringhe nel vettore di hash.

```
1  int main() {
2    memorizza();
3
4    trova("Bari");
5    trova("Bologna");
6    trova(str[0]);
7    trova(str[N1 - 1]);
8
9    return 0;
10 }
```

Il `main` si limita a richiamare la funzione per la memorizzazione delle stringhe e, successivamente, effettua delle chiamate di test alla funzione `trova` per verificare il comportamento dell'algoritmo di ricerca i vari casi. In particolare, la ricerca delle stringhe `Bari` e `Bologna` non avrà sicuramente successo, in quanto le due stringhe non compaiono tra quelle da memorizzare nel vettore di hashing. Vengono poi ricercate la prima e l'ultima stringa inserita, cioè `str[0]` e `str[N1 - 1]`. Per come - si vedrà - è realizzata la funzione di memorizzazione, la seconda stringa viene sicuramente trovata, mentre la prima stringa ha la minima probabilità di essere trovata. Come si è detto, infatti, questo semplice programma non gestisce le collisioni, e un dato inserito più di recente sovrascrive il dato eventualmente presente all'indice corrispondente. Da notare, infine, che la funzione di hash non viene utilizzata direttamente nel `main`, ma soltanto all'interno delle funzioni di memorizzazione e di ricerca.

```
1  int hash(char *s, int n)
2  {
3    int i = 0;
4    int sum = 0;
5
6    do {
7      sum += s[i];
8      i++;
9    } while (s[i] != '\0');
10
11   return sum % n;
12 }
```

Il cuore della tecnica di hashing è contenuto nella funzione `hash`, che calcola l'indice al quale memorizzare o reperire una determinata stringa. La funzione accetta come parametro la stringa sulla base della quale generare l'indice, e il numero `n` di elementi del vettore di hashing. Per calcolare l'indice viene prima calcolata la somma tra tutti i valori numerici dei caratteri ASCII che compongono la stringa, e di questo valore viene calcolato il resto della divisione intera per `n`. In questo modo viene

generato un numero nell'intervallo $[0 \ldots n-1]$ che dipende da tutti i caratteri che compongono la stringa.

La tecnica adottata è una delle tante possibili. In alternativa si potrebbe scegliere di effettuare un'operazione diversa tra i caratteri invece della somma, oppure si potrebbe limitare il numero di caratteri considerati per il calcolo. Quest'ultima soluzione, in particolare, riduce la complessità del calcolo dell'indice. Infine, da sottolineare che questa funzione discrimina tra caratteri maiuscoli e minuscoli.

```
1  int trova(char *s)
2  {
3    int h, ret, comp;
4
5    h = hash(s, N2);
6
7    if (v[h] != NULL) comp = strcmp(s, v[h]);
8    else comp = -1;
9
10   if (!comp) ret = h;
11   else ret = -1;
12
13   printf("[trova] cerca '%s' (pos %d) trovato '%s' ret %d\n",
14          s, h, v[h], ret);
15
16   return ret;
17 }
```

La funzione `trova` ritorna l'indice della stringa `s` all'interno del vettore di hash, e ritorna -1 in caso la stringa non sia presente.

Il valore `h` dell'indice viene ottenuto chiamado `hash`. Se il puntatore che si trova nella posizione corrispondente all'indice ottenuto è nullo la funzione ritorna direttamente -1), altrimenti viene usata la funzione `strcmp` per verificare che la stringa ivi memorizzata sia quella ricercata. Se lo è, cioè `comp` vale 0, a `ret` viene assegnato il valore di `h`, altrimenti -1. La funzione ritorna poi il valore di `ret`, e prima di ritornare effettua una stampa di debug dei vari valori.

```
1  void memorizza()
2  {
3    int i, h;
4
5    for (i = 0; i < N1; i++) {
6      h = hash(str[i], N2);
7
8      printf("(%2d) %s\n", h, str[i]);
9
10     /* fa puntare l'elemento di indice h del vettore v alla
11      * stringa avente l'hash corrispondente; sovrascrive un
12      * eventuale precedente assegnamento */
13     v[h] = str[i];
14   }
15
16   printf("\n");
17 }
```

La funzione `memorizza` inserisce tutte le stringhe contenute nel vettore `str` all'interno del vettore di hashing `v`. L'indice al quale memorizzare il dato è ottenuto tramite la chiamata ad `hash`. Dopo aver stampato delle informazioni di debug, viene effettuato l'assegnamento `v[h]=str[i]`. Da notare che l'assegnamento assegna un puntatore, non effettua la copia fisica della stringa. Infine, è molto importante notare che l'assegnamento viene fatto sempre, a prescindere che il valore eventualmente già presente nel vettore sia nullo o meno. Questo significa che qualsiasi valore precedentemente memorizzato viene perduto. Come corollario, questo significa che gli ultimi valori memorizzati hanno la probabilità maggiore di non venire perduti, ed in particolare questo assicura che l'ultimo valore inserito sarà sicuramente trovato nel vettore di hashing.

19.3 RICERCA E ORDINAMENTO CON LE FUNZION DI LIBRERIA

La libreria standard implementa due funzioni molto utili per la manipolazione di insiemi omogenei di dati: la funzione `bsearch` e la funzione `qsort`, che implementano rispettivamente l'algoritmo di ricerca binaria e l'algoritmo di ordinamento Quick Sort. Entrambi gli algoritmi sono implementati in modo generalizzato, ovvero indipendente dal tipo dei dati da ordinare o ricercare.

La funzione `qsort` è dichiarata come segue:

```
void qsort ( void *base,
             size_t nmemb,
             size_t size,
             compar_fn_t compar );
```

essa effettua l'ordinamento di `nmemb` elementi dell'insieme `base`, ciascuno dei quali ha dimensione `size`; viene usata la funzione `compar` per confrontare due elementi dell'insieme.

La funzione `bsearch`, invece presenta la seguente dichiarazione:

```
void *bsearch ( void *key,
                void *base,
                size_t nmemb,
                size_t size,
                compar_fn_t compar );
```

essa effettua una ricerca binaria dell'elemento `key` all'interno dell'insieme `base`; l'insieme consiste di `nmemb` elementi, ciascuno avente dimensione `size`; viene usata la funzione `compar` per confrontare due elementi dell'insieme.

Essendo funzioni generiche, esse operano allo stesso modo su ogni tipo di insieme, trattando ciascun elemento dell'insieme semplicemente come una sequenza di bit avente una determinata lunghezza. Come si nota, infatti, sia le chiavi che l'insieme di elementi sono specificati mediante puntatori a `void`. Questo permette di passare puntatori a tipi di dati specifici permettendo comunque una dichiarazione generica per l'interfaccia della funzione. Per la stessa ragione, si deve specificare quanti sono gli elementi da ordinare o tra i quali ricercare, e in ciascuna funzione deve essere anche specificata la dimensione del singolo elemento. In particolare, la funzione `qsort` utilizza questa informazione anche per scambiare due valori. Alla funzione non interessa il *significato* dei bit che compongono i valori da scambiare, ma soltanto la loro posizione in memoria e la dimensione.

La genericità della soluzione si può comprendere efficaciemente dalla Figura 19.2, nella quale è rappresentata la porzione di memoria allocata per il vettore da ordinare o sul quale effettuare la ricerca. La memoria è divisa in blocchi di dimensione `size`, non nota a priori, per un totale di `nmemb` elementi a partire dall'indirizzo `base`. Ciascun blocco può quindi immagazzinare indifferentemente un dato semplice di tipo intero, a virgola mobile, oppure un tipo di dato complesso come una struttura dati, o un vettore.

Sia la `bsearch` che la `qsort`, così come tutti gli algoritmi di ordinamento e di ricerca, devono poter confrontare tra loro due valori dell'insieme. Nel caso della ricerca questo serve per decidere

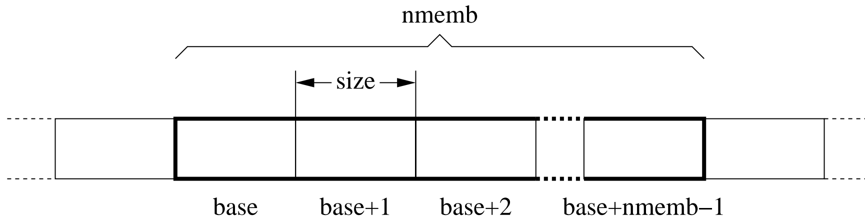


Figura 19.2 Un vettore di `nmemb` elementi di dimensione generica `size` e indirizzo di partenza pari a `base`.

se l'elemento analizzato corrisponde a quello da trovare e, in caso contrario, per procedere opportunamente con la ricerca. Nel caso dell'ordinamento, bisogna stabilire la relazione d'ordine tra due elementi, ovvero stabilire se sono uguali oppure quale dei due è maggiore dell'altro. Dal momento che le funzioni presentate sono generiche, non esiste un unico modo per confrontare due elementi, poiché il confronto dipende dal significato dei bit che compongono gli elementi.

L'ultimo parametro di entrambe le funzioni è quindi un puntatore a funzione, che punta ad una funzione definita dall'utente la quale confronta due elementi dell'insieme. Tale funzione è definita come segue

```
typedef int (*__compar_fn_t) (void *, void *);
```

essa accetta come parametri i puntatori agli elementi da confrontare, e ritorna

- un valore minore di 0 se il primo parametro è minore del secondo;
- 0 se sono uguali;
- un valore maggiore di 0 se il primo parametro è maggiore del secondo.

Per esempio, il confronto tra due interi o tra numeri a virgola mobile può essere fatto banalmente come segue:

```
int compare_double(const void *a, const void *b)
{
  const double *da = (const double *) a;
  const double *db = (const double *) b;

  return (*da < *db) - (*da > *db);
}
```

da notare l'uso della clausola `const` che indica al compilatore che le variabili passare per riferimento non sono modificabili. Questo viene fatto per cautelarsi da eventuali istruzioni errate che vadano a modificare esplicitamente il valore delle variabili.

Ma nel caso fosse necessario confrontare elementi più "complessi", per esempio una struttura composta da vari campi dal significato particolare, deve allora essere il programmatore a fornire una opportuna funzione la quale, dati due elementi, ritorni un codice che indica la relazione d'ordine (l'uguaglianza per la ricerca è un caso particolare).

Il programma completo è contenuto nel file `sort_search.c`.

Data la seguente struttura dati

```
struct t_persona {
  char *nome;
  char *species;
  int anno;
  int mese;
  int giorno;
};
```

ed un vettore di tali strutture, inizializzato per esempio come segue:

```
struct t_persona persona[] = {
  {"Kermit", "Rana", 1975, 6, 3},
  {"Piggy", "Maiale", 1981, 10, 3},
  {"Gonzo", "LaCosa", 1984, 1, 3}
};
```

se si desidera ordinare o ricercare gli elementi del vettore per anno di nascita, bisogna tenere conto che, a parità di anno, bisogna discriminare in base al mese, e a parità di mese bisogna discriminare in base al giorno. Una tale comparazione può essere realizzata con il codice seguente:

```
int cmp_anno(const void *c1p, const void *c2p)
{
  struct t_persona *c1 = (struct t_persona *)(c1p),
                   *c2 = (struct t_persona *)(c2p);
  int ret;

  if (!(ret = ((c1->anno > c2->anno) - (c1->anno < c2->anno))))
    if (!(ret = ((c1->mese > c2->mese) - (c1->mese < c2->mese))))
      ret = ((c1->giorno > c2->giorno) - (c1->giorno < c2->giorno));

  return ret;
}
```

Se invece si desidera comparare, cercare o ordinare, ad esempio sulla base del nome, si utilizzerà semplicemente la funzione `strcmp` di comparazione tra stringhe, come nello spezzone di codice seguente:

```
int cmp_nome(const void *c1p, const void *c2p)
{
  struct t_persona *c1 = (struct t_persona *)(c1p),
                   *c2 = (struct t_persona *)(c2p);
  return strcmp(c1->nome, c2->nome);
}
```

Così facendo i valori sono ordinati in ordine crescente. Per ottenere l'ordinamento in senso decrescente basta banalmente sostituire all'istruzione 5 la seguente:

```
  return -strcmp(c1->nome, c2->nome);
```

nella quale si effettua il cambio di segno del valore ritornato da `strcmp`.

Infine, la chiamata alla funzione di libreria `qsort` per effettuare l'ordinamento per nome è la seguente:

```
qsort(persona, count, sizeof(struct t_persona), cmp_nome);
```

dove `count` indica il numero di elementi contenuti nel vettore, e `cmp_nome` è la funzione che verrà utilizzata da `qsort` per comparare due elementi del vettore.

Capitolo 20

ESERCIZI E ALGORITMI

20.1 PROGRAMMAZIONE IN C

Esercizio 1

Scrivere un programma che stampi la dimensione in byte dei diversi tipi di dati primitivi previsti nel linguaggio C.

Esercizio 2

Scrivere un programma in linguaggio C che stampi i primi n numeri della serie di Fibonacci ($x_1 = x_2 = 1$, $x_n = x_{n-1} + x_{n-2}$ per $n \geq 3$). Il valore di n viene fornito da tastiera. Controllare che valga la condizione $n > 0$.

Esercizio 3

Scrivere un programma che calcoli l'area del cerchio di raggio r.

Si consiglia di definire (con la direttiva `#define`) la costante simbolica PIGRECO da utilizzarsi nel calcolo.

Si usi la funzione matematica `pow(x, y)` che esegue il calcolo x^y. (x e y devono essere dichiarate di tipo `double`). Es. `pow(x, 2.0)` calcola x^2.

Si ricorda che per utilizzare le funzioni matematiche va inserita nel proprio programma la direttiva

```
#include <math.h>
```

e la compilazione va effettuata specificando l'opzione `-lm`. Esempio: `xlc cerchio.c -lm`

Esercizio 4

Date le seguenti dichiarazioni:

```
int a = 0, b = 5, c =3;
```

si valutino le espressioni seguenti verificandone poi il valore con un programma C:

```
a * b % c + 1
++ a - b-- + c
14 - -c * ++a
```

Esercizio 5

Dati tre numeri forniti da tastiera se ne calcoli il massimo.

Esercizio 6

Dati tre numeri a, b, c forniti da tastiera in ordine crescente, verificare che esista un triangolo avente come misura dei lati tali numeri e stabilire di che tipo è il triangolo. Si effettuino opportuni controlli per verificare che i numeri siano stati forniti da tastiera in ordine crescente. (Si ricorda che, dati tre numeri a, b, c ordinati in modo crescente, esiste un triangolo avente come misura dei lati tali numeri se $c < a + b$).

Esercizio 7

Sia dato il sistema lineare a due incognite

$$\begin{cases} ax + by = c \\ a'x + b'y = c' \end{cases}$$

Scrivere un programma che, acquisiti da tastiera i coefficienti e il termine noto delle equazioni, risolva il sistema con il metodo di Cramer.

Metodo di Cramer: siano $Ds = ab' - a'b$, $Dx = cb' - c'b$ e $Dy = ac' - a'c$. Si possono verificare le seguenti situazioni:

1. se $Ds = 0$ e $Dx = 0$ allora il sistema è indeterminato;
2. se $Ds = 0$ e $Dx \neq 0$ allora il sistema è impossibile;
3. se $Ds \neq 0$ le soluzioni del sistema sono:

$$x = \frac{Dx}{Ds} \qquad y = \frac{Dy}{Ds}$$

Esercizio 8

Date le misure dei lati di un rettangolo a, b fornite da tastiera, si scriva un programma che calcoli il perimetro, l'area o la diagonale del rettangolo secondo la richiesta dell'utente. (Si supponga che l'utente possa inserire come scelta: 1 = perimetro, 2 = area, o 3 = diagonale). (Usare l'enunciato switch).

Esercizio 9

Scrivere un programma che legga da tastiera n numeri reali (n richiesto da tastiera), li memorizzi in un vettore (dimensione massima 100 elementi) li stampi a video, ed effettui sui dati i seguenti calcoli:

minimo
massimo
media dei valori
range (massimo-minimo)
stampando a video i risultati ottenuti. (elaborazione di vettori, uso della redirezione dell'input)

Esercizio 10

Scrivere un programma che, attraverso l'uso di funzioni, calcoli l'area del cerchio e la lunghezza di una circonferenza di diametro d.

Esercizio 11

Stabilire se un numero naturale n fornito da tastiera è pari o dispari. Si scriva a tale scopo una funzione chiamata `pari()` che restituisca 1 se il numero è pari e 0 altrimenti e la si richiami dalla funzione principale (`main()`) per stampare l'opportuno messaggio a video.

Esercizio 12

Dato un numero $N > 0$ si calcoli il più piccolo numero M tale che $M! > N$. Si definisca nel programma una funzione che calcoli il fattoriale di un numero.

Esercizio 13

Sommare i primi n numeri naturali (n richiesto da tastiera). Si realizzi l'esercizio implementando due funzioni: una che effettui la lettura di n e l'altra che effettui il calcolo della sommatoria.

Esercizio 14

Sostituire la funzione che calcola la sommatoria nell'esercizio precedente con una funzione che calcoli la produttoria dei primi n numeri naturali.

Esercizio 15

Dati i coefficienti a, b, c di un polinomio di secondo grado $ax^2 + bx + c$, si calcolino e stampino a video i valori assunti dal polinomio per x che varia nell'intervallo $[0, 3]$. La variabile x partirà dal valore 0 e dovrà raggiungere il valore 3 con incrementi di 0.1. Si crei a tal scopo una funzione

```
double f(double a, double b, double c, double x){...}
```

che calcoli il valore di un polinomio arbitrario di grado 2.

Esercizio 16

Si scriva un programma che effettui il calcolo approssimato per $x \to 0$ della funzione $\log(1 + x)$ utilizzando il polinomio:

$$x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots + (-1)^{n+1}\frac{x^n}{n}$$

Il programma dovrà leggere da tastiera le seguenti coppie di informazioni:

- un valore reale che rappresenta la precisione ϵ con cui approssimare il valore della funzione;
- il valore di x in cui calcolare la funzione;

e dovrà stampare a terminale le seguenti informazioni:

- il valore di x in cui è calcolata la funzione;
- la precisione ϵ con cui è calcolata la funzione;
- il valore di n;
- il valore della funzione approssimata;
- il valore vero della funzione.

Esercizio 17

Si scriva un programma che effettui il calcolo approssimato per $x \to 0$ della funzione e^x utilizzando il polinomio:

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

Il programma dovrà leggere da tastiera le seguenti coppie di informazioni:

- un valore reale che rappresenta la precisione ϵ con cui approssimare il valore della funzione;
- il valore di x in cui calcolare la funzione;

e dovrà stampare a terminale le seguenti informazioni:

- il valore di x in cui è calcolata la funzione;
- la precisione ϵ con cui è calcolata la funzione;
- il valore di n;
- il valore della funzione approssimata;
- il valore vero della funzione.

Esercizio 18

Scrivere una funzione di nome `multiplo()` che data una coppia di interi determini se il secondo sia multiplo del primo. La funzione dovrà ricevere due argomenti interi e restituire 1 se il secondo valore è multiplo del primo, 0 in caso contrario. Si utilizzi questa funzione in un programma che acquisisca da tastiera una serie di coppie di interi e che abbia termine quando l'utente intende terminare l'immissione di valori. A tal scopo si effettui un ciclo che ad ogni iterazione legga una coppia di valori usando la funzione `scanf()`. Il ciclo dovrà terminare quando l'utente inserisce da tastiera ^D (in UNIX) o ^Z (in DOS). In questo caso scanf restituisce il valore EOF (-1). (Leggere attentamente sul manuale la documentazione relativa alla funzione `scanf()` ed ai valori che essa restituisce.)

Esercizio 19

Definire la funzione `ipotenusa()` che calcoli la lunghezza dell'ipotenusa di un triangolo rettangolo quando siano dati i due cateti. Si utilizzi questa funzione in un programma che determini la lunghezza dell'ipotenusa per ognuno dei seguenti triangoli. La funzione dovrà ricevere due argomenti di tipo double e restituire la lunghezza dell'ipotenusa come valore `double`.

triangolo	lato 1	lato 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

Esercizio 20

Un garage addebita un importo minimo di $2,00 per un parcheggio fino a tre ore. Il garage addebita un'addizionale di $0,5 per ogni ora o frazione di essa che ecceda le tre di base. L'addebito massimo per ogni dato periodo di 24 ore è di $10,00. Assumere che nessuna auto parcheggi per più di 24 ore per volta.

Si scriva un programma che calcoli e visualizzi gli addebiti per ognuno dei clienti che hanno parcheggiato le loro auto in questo garage. Bisogna immettere da tastiera il numero di ore di parcheggio per ogni cliente. Il programma dovrà visualizzare i risultati in forma tabulare ordinato e dovrà calcolare e visualizzare anche il totale delle ore e degli importi relativi ai clienti. Si scriva a tal scopo un funzione di nome `CalcolaAddebito()` per determinare l'addebito di ogni cliente. I risultati dovranno apparire nel seguente formato:

Cliente	Ore	Addebito
1	1.5	2.00
2	4.0	2.5
3	24.0	10.00
TOT	29.5	14.5

Esercizio 21

Si scriva un programma che memorizzi in una matrice di opportune dimensioni i voti conseguiti durante il primo anno dalle matricole di ingegneria. Si supponga che gli studenti siano al massimo 100 e che gli esami da sostenere durante il primo anno siano 8. I dati vanno forniti nel seguente modo:

- numero studente (intero compreso tra 1 e 100);
- numero esame (intero compreso tra 1 e 8);
- voto esame (intero compreso tra 18 e 30).

Si calcoli infine per ciascun studente il numero di esami superati e la media dei voti e per ciascun esame il numero di voti registrati e la media dei voti.

Per provare il programma si scrivano i dati in un file e si utilizzi la redirezione dell'input. File di esempio:

1	3	23
8	1	29
12	4	25
10	8	30

Esercizio 22

Si consideri un file contenente informazioni relative ai risultati delle partite di calcio di un girone (max 18 squadre). Il file contiene in ciascuna riga, separati da spazio:

- nome della squadra (max 10 caratteri),
- numero di partite vinte,
- numero di partite perse,
- numero di partite pareggiate,
- goal fatti,
- goal subiti.

Si scriva un programma che definita un'opportuna struttura dati, carichi in memoria questi dati e generi un secondo file che contiene in ciascuna riga le seguenti informazioni:

- nome squadra;
- numero partite giocate;
- punteggio (3 punti per partita vinta e 1 per partita pareggiata);
- media dei goal fatti per partita;
- media dei goal subiti per partita.

Esempio di file dati:

AAAAAA	3	1	1	4	2
BBBBBB	2	1	2	2	3
CCCCCC	4	1	0	5	1
DDDDDD	2	0	2	2	2

Esercizio 23

Data la seguente porzione di codice in linguaggio C, si dica quali valori verranno stampati ad ogni iterazione.

```
#include <stdio.h>

#define   MAX   10

int main(void)
{
int f0 = 0, f1 = 1, n, temp;

   for (n = 2; n <= MAX; ++n) {
      temp = f1;
      f1 += f0;
      f0 = temp;
      printf("%d %d\n", n, f1);
   }
}
```

Esercizio 24

Data la seguente porzione di codice in linguaggio C, si costruisca il flowchart corrispondente e si dica quali strutture di controllo sono state utilizzate per rappresentarlo.

```
int i, x=5;
float a;
for (i=100; i>0; i-=2){
   x /= i;
   if(i >= 50  && i <= 80 )
     a = x/3;
   else
     a = x/2;
}
```

Esercizio 25

Data la seguente porzione di codice in linguaggio C, si dica il valore che assume la variabile j durante la prima iterazione e il valore della variabile primo alla fine dell'esecuzione del programma.

```
int main(void)
{
    int primo = 1;
    int i = 2, j, x = 5;

    while ((i < x) && primo)
      {
        j = x % i;

        if (j == 0)
            primo = 0;
        else
            i++;
      }
}
```

Esercizio 26

Data la seguente porzione di codice in linguaggio C, si costruisca il flowchart corrispondente e si dica quali strutture di controllo si sono utilizzate per rappresentarlo.

```
int primo = 1;
int i = 2, j, x=5;

while ((i < x) && primo)
{
  j = x % i;

  if (j == 0)
    primo = 0;
  else
    i++;
```

}

Esercizio 27

Un'azienda retribuisce i suoi venditori con delle provvigioni. Un venditore riceve $200 alla settimana più il 9% delle proprie vendite lorde portate a termine in quella settimana. Per esempio, un venditore che faccia incassare $3000 di venduto lordo, in una settimana, riceverà $200 più il 9% di $3000, cioè un totale di $470. Scrivere un programma, utilizzando un vettore di contatori, che determini quanti venditori abbiano guadagnato una retribuzione compresa in ognuno dei seguenti intervalli (si tronchi la retribuzione a una somma intera):

1	$200-$399
2	$400-$599
3	$600-$799
4	$800-$999
5	$1000 e oltre

Esercizio 28

Scrivere un programma che simuli il lancio di due dadi. Il programma deve utilizzare la funzione `rand()` per lanciare il primo dado e invocarla nuovamente per lanciare il secondo dado (si veda la descrizione della funzione sul manuale). Quindi dovrà essere calcolata la somma dei due valori. Il programma dovrà lanciare i dadi 36.000 volte. Si utilizzi un vettore unidimensionale per contare il numero di occorrenze di ogni somma possibile (cioè per tutti i valori compresi tra 2 e 12). Si visualizzi alla fine una tabella che riporti per ogni possibile valore la relativa frequenza.

Nota: poichÃ© ogni dado può mostrare un valore intero compreso tra 1 e 6, la somma dei due valori sarà compresa tra 2 e 12 e il vettore di contatori dovrà avere dimensione 11 (ogni elemento del vettore deve contare il numero di occorrenze di uno degli 11 valori possibili).

Esercizio 29

Una piccola compagnia aerea ha appena comprato un computer per il suo nuovo sistema di prenotazione automatica. Scrivere un programma che assegni i posti su ogni volo dell'unico aereo dell'aerolinea. (capacità 10 posti).

Il programma dovrà visualizzare il seguente menu di scelte:

1. Inserisci 1 per posto fumatori
2. Inserisci 2 per posto non fumatori

Nel caso in cui il cliente scelga 1 il programma dovrà assegnare uno dei cinque posti (da 1 a 5) nella sezione fumatori. Nel caso che il cliente digiti 2, allora il programma dovrà assegnare un posto nella sezione non fumatori (da 6 a 10). Il programma dovrà infine stampare a video la situazione di prenotazione di tutti i posti, con l'indicazione della tipologia di posto (fumatori/non fumatori).

Si utilizzi un vettore unidimensionale per rappresentare la mappa dei posti sull'aereo. Si azzerino tutti gli elementi del vettore in modo da indicare che tutti i posti sono liberi. Man mano che i posti verranno assegnati si dovrà impostare a 1 l'elemento corrispondente del vettore in modo da indicare che il posto non è più disponibile.

Quando la sezione richiesta dal cliente è piena si deve richiedere al cliente se sia disposto ad accettare una sistemazione nell'altra sezione.

Esercizio 30

Scrivere un programma che prenda in input quattro stringhe che rappresentino degli interi, le converta in interi, sommi i valori ottenuti e visualizzi i loro totali. Si usi la funzione `sscanf()` della libreria standard.

Esercizio 31

Scrivere un programma che acquisisca due stringhe da linea di comando le confronti con la funzione `strcmp()` e stabilisca quale delle due precede in ordine alfabetico l'altra e lo comunichi all'utente.

Esercizio 32

Scrivere un programma che legga da tastiera una sequenza di caratteri utilizzando la funzione `getchar()` e la memorizzi come stringa in un vettore. (Si ricordi di terminare la stringa con il carattere "\0".)

Il programma dovrà controllare la fine dell'input della sequenza di caratteri immessa da tastiera attraverso il valore restituito dalla funzione `getchar()` (si veda sul manuale la descrizione della funzione). Se l'utente immette un numero di caratteri che supera la dimensione del vettore allocato, si termini la lettura da tastiere e si avvisi con un messaggio l'utente.

Il programma dovrà dapprima stampare a video la stringa, poi la deve convertire e stampare in maiuscolo e in minuscolo. Si scrivano a tal scopo due funzioni (una di conversione della stringa in maiuscolo e l'altra in minuscolo) che utilizzino le funzioni `tolower()` e `toupper()` della libreria standard.

Esercizio 33

Individuare e correggere l'errore (o gli errori) nel seguente codice C:

```
#include <stdio.h>

int main()
{
  int default = 0;

  printf("introdurre il valore di default: ");
  scanf("%d", &default);

  printf("il valore inserito e' %d\n: ", default);

  return 0;
}
```

20.2 REALIZZAZIONE DI ALGORITMI

1. Dati due numeri interi A e B con $A < B$, calcolarne la somma incrementando A di 1 per B volte.
2. Stampare i primi M numeri naturali successivi a N.
3. Dati tre numeri interi stampare il più grande.
4. Dato un numero $N > 0$ si calcoli il più piccolo numero M tale che $M! > N$.
5. Dati due numeri N ed M positivi con $N > M$, si stampino gli interi compresi tra N ed M estremi inclusi.

6. Dati due numeri interi A e B con $A < B$, calcolarne la somma incrementando A di 1 per B volte.
7. Stampare i primi M numeri naturali successivi a N.
8. Dati due numeri interi stampare il più grande.
9. Dati tre numeri interi stampare il più grande.
10. Dati tre numeri A, B e C, calcolare $A \cdot B \cdot C$ usando solo l'operazione di somma.
11. Dati due numeri X e N calcolare X^N usando solo operazioni di somma.
12. Dato un numero $N > 0$ si calcoli il più piccolo numero M tale che $M! > N$.
13. Dati due numeri N ed M positivi con $N > M$, si stampino gli interi compresi tra N ed M estremi inclusi.

Bibliografia

1. The GNU project. Bash reference manual. *http://www.gnu.org/software/bash/manual/bashref.html*, ultimo accesso: 13 ottobre 2008.
2. Alessandro Rubini. Dispense di C online. *http://gnudd.com/srt-2007/A-C-X.html*, 2007.
3. Alessandro Rubini. Dispense di C online. *http://gnudd.com/srt-2007/A-C-X-more.html*, 2007.
4. Wikipedia. Definizione di filesystem. *http://en.wikipedia.org/wiki/File_system*, ultimo accesso: 29 giugno 2009.

Appendice A

Tabella degli operatori

*

Tabella A.1 Tutti gli operatori del C.

<table>
<tr><th>Symbol</th><th>Name or Meaning</th><th>Associativity</th></tr>
<tr><td colspan="3">Highest Precedence</td></tr>
<tr><td>( )</td><td>Function call</td><td>⟹</td></tr>
<tr><td>[ ]</td><td>Array element</td><td></td></tr>
<tr><td>.</td><td>Structure or union member</td><td></td></tr>
<tr><td>− ></td><td>Pointer to structure member</td><td></td></tr>
<tr><td>!</td><td>Logical NOT</td><td>⟸</td></tr>
<tr><td>~</td><td>One's complement</td><td></td></tr>
<tr><td>-</td><td>Unary minus</td><td></td></tr>
<tr><td>++</td><td>Increment</td><td></td></tr>
<tr><td>–</td><td>Decrement</td><td></td></tr>
<tr><td>&</td><td>Address</td><td></td></tr>
<tr><td>*</td><td>Indirection</td><td></td></tr>
<tr><td>(type)</td><td>Type cast [for example, (float) i]</td><td></td></tr>
<tr><td>sizeof</td><td>Size in bytes</td><td></td></tr>
<tr><td>*</td><td>Multiply</td><td>⟹</td></tr>
<tr><td>/</td><td>Divide</td><td></td></tr>
<tr><td>%</td><td>Remainder</td><td></td></tr>
<tr><td>+</td><td>Add</td><td>⟹</td></tr>
<tr><td>-</td><td>Subtract</td><td></td></tr>
<tr><td><<</td><td>Left shift</td><td>⟹</td></tr>
<tr><td>>></td><td>Right shift</td><td></td></tr>
<tr><td><</td><td>Less than</td><td>⟹</td></tr>
<tr><td><=</td><td>Less than or equal to</td><td></td></tr>
<tr><td>></td><td>Greater than</td><td></td></tr>
<tr><td>>=</td><td>Greater than or equal to</td><td></td></tr>
<tr><td>==</td><td>Equal</td><td>⟹</td></tr>
<tr><td>!=</td><td>Not equal</td><td></td></tr>
<tr><td>&</td><td>Bitwise AND</td><td>⟹</td></tr>
<tr><td>^</td><td>Bitwise exclusive OR</td><td>⟹</td></tr>
<tr><td>|</td><td>Bitwise OR</td><td>⟹</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>⟹</td></tr>
<tr><td>||</td><td>Logical OR</td><td>⟹</td></tr>
<tr><td>? :</td><td>Conditional</td><td>⟸</td></tr>
<tr><td>=</td><td>Assignment</td><td>⟸</td></tr>
<tr><td>*=, /=,
%=, +=,
-=, <<=, >>=,
&=, ^ =, | =</td><td>Compound assignment</td><td></td></tr>
<tr><td>,</td><td>Comma</td><td>⟹</td></tr>
<tr><td colspan="3">Lowest Precedence</td></tr>
</table>

Appendice B

Il compilatore gcc

Esistono vari compilatori C disponibili sia open che closed source, sia gratuiti che a pagamento. Inoltre sono disponibili per pressoché ogni architettura e per ogni sistema operativo.

In questo capitolo verrà introdotto brevemente il compilatore gcc, realizzato nel contesto del progetto GNU.

La scelta di questo compilatore dipende dal fatto che è molto diffuso, è software libero e quindi, tra gli altri vantaggi, è liberamente scaricabile ed utilizzabile per le esercitazioni e per la realizzazione di programmi anche molto complessi. Si pensi per esempio che il kernel di Linux è scritto in C e compilato utilizzando il compilatore gcc.

E' disponibile in tutte le distribuzioni di Linux, ed esistono varie versioni portate su altri sistemi operativi.

B.1 OPZIONI PIÙ IMPORTANTI

Il compilatore gcc, come ogni implementazione di cc, riceve opzioni sulla riga di comando.

I file vengono elaborati in base al proprio nome, ovvero:

- se terminano in .c vengono compilati;
- se terminano in .S vengono solo passati all'assemblatore;
- se terminano in .o vengono solo passati al linker.

*

Tabella B.1 Le opzioni più utilizzate per il compilatore gcc.

gcc opzioni -o file	l'output viene scritto nel file specificato, prevaricando il comportamento predefinito
gcc -c file	*compile only*, il risultato è un file oggetto il cui nome è derivato dal nome del sorgente, anche se di solito si usa -o
gcc -E file	solo preprocessore, il risultato viene scritto su stdout, se non viene specificato -o
gcc -Dsimbolo	definisce la macro di preprocessore, assegnando la stringa vuota
gcc -Dsimbolo=valore	definisce la macro di preprocessore al valore indicato
gcc -Idirectory	specifica di usare la directory indicata nella ricerca dei file di intestazione
gcc -Ldirectory	specifica di usare la directory indicata nella ricerca dei file di libreria
gcc -lnome	specifica di usare la libreria indicata nella fase di link

Le opzioni più importanti del gcc sono riportate in Tabella B.1. In tabella, "file" indica un nome di file, ogni volta diverso.

Un esempio dell'utilizzo di gcc è il seguente:

```
gcc -DDEBUG jpegdemo.c -I/usr/local/include -L/usr/local/lib -ljpeg -o jpegdemo
```

...