Dalla A alla Z passando per C

Tullio Facchinetti
Alessandro Rubini

10 maggio 2009

Indice

1 Ambiente di programmazione **10**
1.1 L'autenticazione 10
1.2 L'interprete dei comandi: la shell 11
1.3 Il file system 12
1.4 Le directory 13
1.4.1 La home directory 14
1.4.2 Le directory . e 14
1.4.3 Muoversi tra directory 15
1.4.4 Elencare il contenuto di una directory 15
1.4.5 Creare e cancellare directory 15
1.4.6 Copiare file: cp 16
1.4.7 Spostare i file: `mv` 17
1.4.8 Cancellare file: `rm` 18
1.5 Le wildcard 19
1.6 Visualizzazione del manuale 20
1.7 Esecuzione di comandi 20
1.8 Elenco riassuntivo dei comandi principali 21

2 Dal problema al programma, passando dall'algoritmo **23**
2.1 Esempi di problemi 23
2.2 La programmazione 24
2.3 L'algoritmo 24
2.4 Proprietà di un algoritmo 25
2.5 Esempi di algoritmo 25
2.5.1 Prodotto di matrici 26
2.5.2 Calcolo del massimo comune divisore 26
2.6 Il programma 26
2.7 Correttezza di un programma 27
2.8 Esecuzione del programma 27
2.9 Errori e debug 27
2.10 Testing 28
2.11 Manutenzione 28
2.12 Linguaggi di programmazione 28
2.13 Evoluzione del linguaggio C 29

3 Realizzazione di un programma 30
3.1 Il file sorgente 30
3.2 Il compilatore 32
3.3 Il linker e file oggetto 32
3.4 Il loader 32
3.5 Strutturazione del codice 33

4 Concetti di base 34
4.1 Il primo programma in C 34
4.2 Compilazione del programma 35
4.3 I commenti 36
4.4 Nota stilistica 36
4.5 Gli identificatori 37
4.6 Le parole chiave 37
4.7 Le variabili 38
4.8 Visualizzazione a video con `printf` 39
4.9 Lettura di dati da tastiera 39
4.10 Specifica di formato per `printf` e `scanf` 41

5 Istruzioni e strutture di controllo 42
5.1 Istruzioni composte 42
5.2 Il costrutto `while` 43
5.3 Il costrutto `do-while` 44
5.4 Il costrutto `for` 45
5.5 Il costrutto `if` 47
5.6 Il costrutto `switch` 49
5.7 Le istruzioni `break` e `continue` 51
5.8 Il costrutto `goto` 52

6 Gli operatori 53
6.1 Precedenza degli operatori 53
6.2 Ordine di valutazione 54
6.3 Il concetto di side effect 54
6.4 Side effect e ordine di valutazione 54
6.5 Associatività degli operatori 55
6.6 Chiamata a funzione 56
6.7 Elemento di vettore 56
6.8 Elemento di struttura 57
6.9 Elemento di struttura da puntatore 57
6.10 Negazione logica 58
6.11 Complemento a 1 59
6.12 Negazione unaria 59
6.13 Incremento e decremento 60
6.14 Estrazione di un indirizzo 61
6.15 Uso di puntatore 61
6.16 Operatore di casting 62
6.17 Dimensione di una variabile 62
6.18 Moltiplicazione e divisione 63
6.19 Resto di divisione intera 64

6.20 Somma e sottrazione 64
6.21 Spostamento dei bit (shift) 65
6.22 Confronto 67
6.23 Confronto: uguaglianza e diversità 68
6.24 AND bit-a-bit 69
6.25 XOR bit-a-bit 69
6.26 OR bit-a-bit 70
6.27 AND logico 70
6.28 OR logico 71
6.29 Espressione condizionale 72
6.30 Assegnamento 73
6.31 Forme abbreviate di assegnamento 74
6.32 Operatore virgola 74

7 Tipi di dati 76
7.1 Tipi interi 76
7.1.1 Conversione da esadecimale a decimale 79
7.2 Tipi a virgola mobile 79
7.3 I puntatori 81
7.4 I vettori 81
7.5 Strutture dati 82
7.5.1 Altri esempi di uso delle strutture 84
7.6 Union 85
7.7 Interi indipendenti dalla piattaforma 85
7.8 Conversioni di tipo 86
7.9 Assegnare nuovi nomi ai tipi di dato: `typedef` 86

8 I puntatori 88
8.1 Puntatori e vettori 89
8.2 L'operatore `sizeof` 91
8.3 Le stringhe 91
8.4 Puntatori e strutture 93
8.5 Argomenti del programma 94
8.6 Puntatori a funzione 94

9 Funzioni 95
9.1 Dichiarazione di funzioni 95
9.2 Definizione di funzioni 96
9.3 Passaggio dei parametri 96
9.4 Passaggio per riferimento 97
9.5 La funzione `main` 99
9.6 Numero variabile di parametri 100

10 Classi di memoria 101
10.1 La visibilità (scope) di dati e funzioni 101
10.2 Classi di memorizzazione 102
10.3 Allocazione di memoria 104
10.4 Variabili auto 105
10.5 Variabili `register` 105
10.6 Variabili `static` 106

10.7 Variabili extern 107
10.8 Inizializzazioni 107

11 Il preprocessore **109**
11.1 La direttiva #define 110
11.2 La direttiva #include 110
11.3 La direttiva #if e #ifdef 111

12 I file **114**
12.1 File binari e file di testo 114
12.2 Accesso a file 115
12.3 Apertura e chiusura di file 115
12.3.1 Apertura di file 116
12.3.2 Chiusura di file 117
12.4 Forzare la scrittura dei dati con `fflush` 117
12.5 Accesso a file binari: le funzioni `fread` e `fwrite` 117
12.6 Lettura di file di testo 119
12.7 Lettura di file strutturati con `fgets` e `sscanf` 121
12.8 I/O con le funzioni `fscanf` e `fprintf` 121
12.9 Esempio di lettura di matrici con `fgets` e `sscanf` 122
12.10 Redirezione dell'input e dell'output 124
12.10.1 Il programma `count.c` 124
12.10.2 Un altro esempio di redirezione da linea di comando 126
12.11 `fscanf` e `fgets` a confronto 127
12.11.1 La bufferizzazione dell'input 127
12.11.2 Il problema del bubber overflow 129

13 Utilizzo di più file sorgente **131**
13.1 Il comando `make` 133
13.2 Esempio di dipendenze 134
13.3 Il makefile 134

14 Le librerie **136**
14.1 Uso di librerie esterne 136
14.1.1 Il comando `ar` 137
14.1.2 Esempio di utilizzo del comando `ar` 138
14.2 La libreria di input/output `stdio.h` 139
14.2.1 Printf 139
14.3 La libreria standard `stdlib` 140
14.3.1 Ricerca e ordinamento 140
14.3.2 Controllo dell'esecuzione del programma 141
14.3.3 Gestione della memoria 142
14.3.4 Generazione di numeri casuali 142
14.4 Manipolazione di stringhe 144
14.5 La libreria matematica 145

15 Stile di programmazione **147**

16 Tecniche di programmazione **149**
16.1 La ricorsione 149

17 Strutture informative 151
17.1 Classificazione delle strutture di dati . . . 151
17.2 Strutture astratte di dati . . . 152
17.3 La lista lineare . . . 153
17.4 La coda . . . 154
17.5 La pila (stack) . . . 154
17.6 La doppia coda . . . 155
17.7 Gli array . . . 156
17.8 Le tavole (tabelle) . . . 156
17.9 I grafi . . . 156
17.10 Gli alberi . . . 158
17.10.1 Visita degli alberi . . . 159
17.10.2 Visita in ordine anticipato . . . 159
17.10.3 Visita in ordine differito . . . 160
17.11 Alberi binari . . . 160
17.11.1 Visita in ordine simmetrico . . . 161

18 Strutture concrete di dati 164
18.1 Struttura sequenziale . . . 164
18.2 Esempio: la coda circolare . . . 167
18.3 Catena o lista . . . 167
18.3.1 Inserimento ed eliminazione di elementi . . . 168
18.4 Memorizzazione delle strutture astratte: liste . . . 170
18.5 Memorizzazione delle strutture astratte: code, pile, doppie code . . . 170
18.6 Memorizzazione delle strutture astratte: matrici . . . 170
18.7 Memorizzazione delle strutture astratte: tavole . . . 171
18.8 Ricerca sequenziale . . . 171
18.9 Ricerca binaria . . . 171
18.10 Accesso diretto . . . 172
18.11 Accesso calcolato (hashing) . . . 172
18.12 Memorizzazione di alberi e grafi in catene . . . 172
18.13 Memorizzazione di alberi e grafi in plessi . . . 172

19 Tecniche di programmazione e algoritmi base 174
19.1 Algoritmi di ricerca . . . 174
19.1.1 La ricerca sequenziale . . . 174
19.1.2 La ricerca binaria . . . 175
19.2 L'ordinamento . . . 177
19.2.1 Bubblesort . . . 177

20 Esercizi e algoritmi 178
20.1 Programmazione in C . . . 178
20.2 Realizzazione di algoritmi . . . 187

A Tabella degli operatori 188

B Il compilatore gcc 190
B.1 Opzioni più importanti . . . 190

Elenco delle figure

1.1 Esempio di albero delle directory. 13

3.1 Schema descrittivo della realizzazione di un programma. 31

4.1 Somma di due numeri letti da tastiera. 40

5.1 Istruzioni composte. 43
5.2 Il costrutto while. 43
5.3 Il costrutto do-while. 45
5.4 Il costrutto for. 46
5.5 Il costrutto if. 48
5.6 Il costrutto if-else. 48
5.7 Costrutto `if` per la determinazione del massimo tra due numeri. 49

7.1 Tabella ASCII. 78

8.1 Una variabile in memoria. 88

9.1 Esempio di passaggio dei parametri. 97

12.1 Il programma `voti.c`. 120

14.1 Simulazione del lancio di due dadi. 143

17.1 Le operazioni che si possono attuare su una coda. 154
17.2 Push e pop: le principali operazioni che si compiono su uno stack. 155
17.3 Due differenti implementazioni di uno stack. 155
17.4 Le operazioni che si possono attuare su una doppia coda. 156
17.5 Esempio di grafo connesso composto da 9 nodi. 157
17.6 Esempio di grafo diretto composto da 9 nodi. 157
17.7 Esempio di albero. 158
17.8 Visita anticipata dell'albero di Figura 17.7. 160
17.9 Visita differita dell'albero di Figura 17.7. 161
17.10Esempio di albero binario. 162
17.11Visita in ordine simmetrico dell'albero binario di Figura 17.10. 162

18.1 Esempio di memorizzazione di una struttura sequenziale. 165
18.2 Esempio di memorizzazione di una struttura sequenziale con elementi di dimensione variabile.166
18.3 Esempio di lista. 167
18.4 Esempio di allocazione in memoria della lista. 167

18.5 Esempio di lista bidirezionale. 168
18.6 Esempio di lista circolare. 168
18.7 Inserimento dell'elemento H_{new} in una lista. 169
18.8 Eliminazione dell'elemento H_i dalla lista. 169
18.9 Grafo di esempio. 173
18.10Esempio di memorizzazione del grafo di Figura 18.9 (sono visualizzati soltanto una parte dei collegamenti).

19.1 Esempio di ricerca binaria. 176

Elenco delle tabelle

4.1 Tabella degli specificatori di formato. 41

7.1 I tipi interi previsti nel linguaggio C. 77
7.2 Intervalli dei valori possibili per i diversi tipi interi del C. 77
7.3 I tipi a virgola mobile previsti nel linguaggio C. 80

8.1 Lo stato rappresentato in Figura 8.1.. 89

10.1 Scope e lifetime delle variabili. 103

11.1 Alcuni header standard del C. 111

A.1 Tutti gli operatori del C. 189

B.1 Le opzioni più utilizzate per il compilatore gcc. 191

Prefazione

Il materiale contenuto in questo documento 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 e Prof.ssa Cristiana Larizza.

Parte del contenuto è una rielaborazione delle dispense redatte da Alessandro Rubini per il corso di Sistemi Real-time presso l'Università di Pavia e disponibile online [Rub07a, Rub07b]. Se il documento di Rubini è "dalla A alla X", questo si propone di essere "dalla A alla Z", quindi una sostanziale estensione della versione originale di Rubini.

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

Capitolo 1

Ambiente di programmazione

PER POTER PROGRAMMARE in C sono necessari una serie di strumenti software che bisogna 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, con cui per esempio 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. 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, o *username*, o *user id*. Lo username viene assegnato dall'amministratore del sistema, e con tale nome si viene univocamente identificati dal sistema. Il sistema risponde chiedendo

```
Password:
```

Si noti che:

- i caratteri battuti come password sono invisibili;
- se la password è corretta, il sistema mostra il prompt.

Errori su username o password vengono notificati. Per esempio:

```
login: utenet1
Password:
Login incorrect
```

Un esempio di login è il seguente:

```
login: user007
Password: pl67kf2
```

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 come la compilazione, l'esecuzione dei programmi, 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.

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

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

dove

user1	è lo username
europa	è il nome del computer in uso
~	è la directory corrente (vedi Sezione 1.4).
$	è un carattere convenzionale per delimitare il prompt dall'input dell'utente

Il prompt non viene proposto da Unix direttamente, ma da un programma detto *shell* che, come quelli che possono essere scritti dagli utenti, gira su Unix. La shell si occupa di interpretare i comandi inseriti dall'utente e di eseguire le azioni corrispondenti. E' così chiamata in quanto costituisce la conchiglia (shell in inglese) più esterna del sistema Unix. Quest'ultimo può infatti essere pensato come una serie di livelli logici tra la macchina e l'utente, il più esterno dei quali, quello più vicino all'utente, è appunto la shell. [pro08] fornisce una trattazione molto completa della shell è disponibile.

I caratteri battuti sulla tastiera 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 Il file system

Ogni file ha un nome che può essere composto dai seguenti caratteri:

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

Si tenga presente che Unix distingue tra lettere maiuscole e minuscole. Alcuni esempi di nomi di file sono: `lezione.doc`, `lezione.old`, `file_mio`, `19.nov.92`

Un modo semplice per mettere nel file `agenda` i byte che rappresentano i caratteri `pranzo di lavoro` è:

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

Il comando `echo` ripete quello che gli si dice sulla riga di comando. In genere, tale ripetizione porta alla visualizzazione del contenuto della riga di comando sullo schermo. Infatti 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.

Ciò che viene effettuato 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 eliminato. 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)catenate*, 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. E' possibile appendere del contenuto ad un file esistente con il comando:

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

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 usasse 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

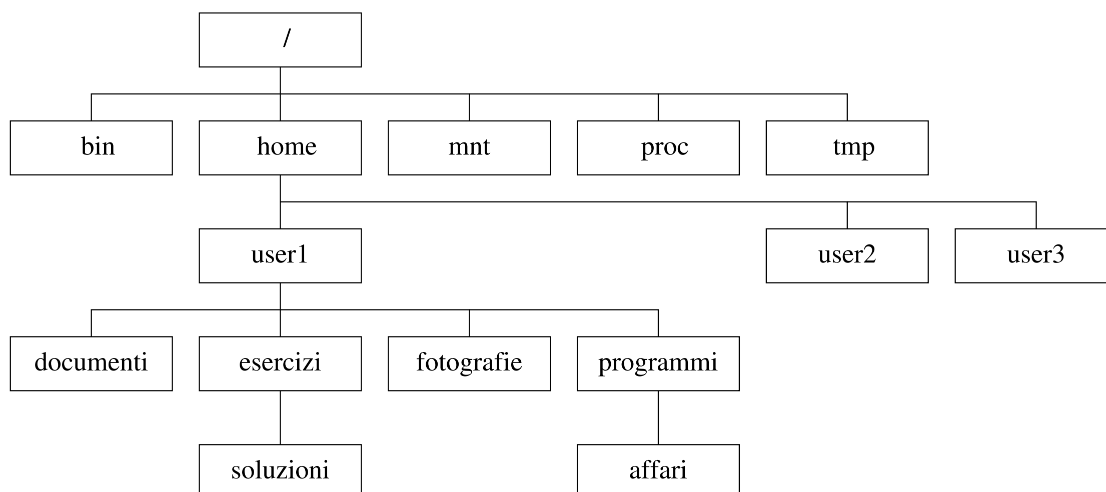

Figura 1.1: Esempio di albero delle directory.

```
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`.

1.4 Le directory

Le *directory* servono per organizzare in modo "gerarchico" i file, in una struttura che formalmente si chiama *grafo aciclico*, e che 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 / (slash).

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 `user1`, in quanto directory diverse possono avere lo stesso nome, qualora siano contenute 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 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/`

NOTA Un percorso si dice assoluto se inizia per /, altrimenti si dice relativo; nei nomi il carattere / ha 2 usi: (1) indica la directory `root` e (2) funge da separatore di directory.

1.4.1 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*. 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. Il carattere ~ può essere usato come sostitutivo del percorso assoluto di una directory. Per esempio, le seguenti forme sono del tutto valide:

```
~/programmi
~/..
```

e vengono interpretate rispettivamente come

```
/home/user1/programmi
/home
```

(il significato della directory `..` viene illustato in Sezione 1.4).

1.4.2 Le directory . e ..

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

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

L'esempio sotto va interpretato tenendo presente l'albero delle directory di Figura 1.1:

```
user1@europa:~$ ls -C
documenti  esercizi  fotografie  programmi
user1@europa:~$ ls -C .
documenti  esercizi  fotografie  programmi
user1@europa:~$ cd ese*/solu*
user1@europa:~/esercizi/soluzioni$ ls -C ../..
documenti  esercizi  fotografie  programmi
user1@europa:~/esercizi/soluzioni$ cd ../soluzioni
user1@europa:~/esercizi/soluzioni$ cd ../soluzioni/../../../user1
user1@europa:~$
```

1.4.3 Muoversi tra directory

Il comando `cd` serve a spostarsi tra le directory. Per esempio

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

Il comando `cd`, a differenza di altri comandi come `mkdir` e `rmdir`, non è implementato come un programma presente sul disco e chiamato `cd`, ma è un comando interno alla shell[1].

Il comando `pwd` scrive sul suo standard output la directory corrente. Ad esempio:

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

Anche il comando `pwd` è un comando interno alla shell.

1.4.4 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 opzioni più importanti e utilizzate sono:

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

1.4.5 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
```

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

```
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 `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!). Con `rmdir tmp` si è tentato di cancellare la directory `tmp` che però non è vuota, e quindi non si può cancellare. Infine, con `ls -l tmp/d1` si è tentato di listare il contenuto della directory `tmp/d1`, che era stata precedentemente cancellata e quindi non esisteva 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. E' comunque possibile redirigere anche lo standard error, anche se ciò.

1.4.6 Copiare file: cp

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 `dir`. Valgono le seguenti regole:

- `dir` deve esistere come directory
- se `f1` esiste già dentro `dir`, il file viene sovrascritto

Il comando

```
cp f1 f2
```

crea una copia del file `f1` di nome `f2`. 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), poichè contiene anche l'ultima stringa inserita.

E' 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:~$
```

1.4.7 Spostare i file: mv

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

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

il quale sposta `f1...` dentro `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:~$
```

1.4.8 Cancellare file: **`rm`**

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

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

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

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

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

```
rm -r dir
```

cancella ricorsivamente la directory `dir` insieme con tutte le subdirectory e tutti i file contenuti.

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

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

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.

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

Su molte macchine l'amministratore ha deciso che `rm` e' equivalente a `rm -i`, nel qual caso viene sempre chiesta conferma quando si cancellano i file. L'argomento `-f` ("force") permette di imporre la cancellazione (o la sovrascrittura) senza una richiesta di conferma. Quindi il comando

```
rm -fr dir
```

elimina la directory `dir` e tutto il suo contenuto senza chiedere alcuna conferma all'utente.

1.5 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.
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.

1.6 Visualizzazione del manuale

E' 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`, che non è altro che la shell stessa.

1.7 Esecuzione di comandi

I comandi disponibili in un sistema UNIX non sono altro che dei programmi eseguibili fisicamente presenti sul disco che vengono lanciati 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 visti 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 comnadi che vengono invocati senza specificare il percorso[2]. Non ci soffermeremo su come questa opportunità può essere sfruttata.

E' però importante sottolineare la differenza tra due diverse linee di comando per l'invocazione programmi, che talvolta vengono confuse. Per esempio:

```
$ ../rm nomefile
```

```
$ rm ../nomefile
```

Il primo comando invoca il programma `rm` presente nella directory `..` della directory corrente[3], 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 programma.

1.8 Elenco riassuntivo dei comandi principali

In questa sezione verranno riassunti brevemente alcuni dei comandi più comuni da utilizzare con l'inteprete di comandi di Unix, e che si riveleranno utili per gestire lo sviluppo dei programmi in C.

Per una descrizione sommaria che comprende un maggior numero di comandi, si rimanda a [Amb08].

`cat filename` — visualizza il contenuto di un file

`cd dir` — cambia la directory corrente spostandosi in `dir`
```
cd newdir
cd /home/user/newdir
```

`man comando` — mostra il manuale con tutte le informazioni relative al comando `comando`
```
man cat
man man
```

`echo testo` — visualizza sul video la stringa `testo`; il suo utilizzo si rivela molto comodo in abbinamento agli operatori di redirezione; per esempio
`echo testo > file` inserisce `testo` nel file `file`

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

[3] 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.

`mkdir dir` crea una nuova directory chiamata `dir` come sottodirectory della directory corrente

`rm file` cancella il file di nome `file`
`rm prova.c` cancella il file `prova.c`
`rm -fr elem`] cancella `elem` forzatamente, senza chiedere conferma; se `elem` è una directory, la cancella ricorsivamente, cancellando cioè tutti i file e le sottodirectory contenute

`rmdir dir` cancella la directory `dir`; il comando funziona soltanto se `dir` è completamente vuota

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

La descrizione del problema non fornisce (in genere) 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 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. interpretate ed eseguite dalla macchina e le corrispondenti operazioni vengono realizzate. Il *processore*, il componente più importante di un sistema di calcolo, si occupa proprio di eseguire le istruzioni che compongono il programma.

2.3 L'algoritmo

Per risolvere un problema bisogna definire (ovvero, identificare o progettare) un procedimento risolutivo, ossia un insieme di passi elementari (istruzioni) che, eseguiti secondo un ordine prestabilito, permettono di arrivare ai risultati 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 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 (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.

2.4 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 richiede un uso limitato di risorse

Tipiche risorse che devono essere salvaguardate sono il tempo di esecuzione e 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*

2.5 Esempi di algoritmo

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

2.5.1 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

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

2.5.2 Calcolo del massimo comune divisore

Problema: calcolare il massimo comune divisore di due numeri interi.
Dati di ingresso: due numeri interi a e b.
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*.

2.6 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 per l'implementazione di un programma sono 4:

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. E' 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 poichè 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 in questo caso 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 interpretate e le corrispondenti operazioni vengono eseguite.

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

- caricamento in memoria, tipicamente a partire da una periferica di memoria di massa, come un disco rigido
- identificazione del "punto d'ingresso" del programma
- esecuzione sequenziale delle istruzioni (fetch + esecuzione)

2.9 Errori e debug

Durante la scrittura del codice che costituisce un programma, capita invariabilmente 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. 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, viene collaudato per verificarne la correttezza semantica, 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.

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 del 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.

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 s/w), 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. Non appena viene caricato nella memoria del computer dal sistema operativo, il processore può iniziare ad interpretare le istruzioni e, quindi, ad eseguire il programma.

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 sia piuttosto generale, ed può essere quindi ritenuta valida per qualsiasi linguaggio di programmazione, lo schema di Figura 3.1 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 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 decodifica 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 un programma Java.

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.

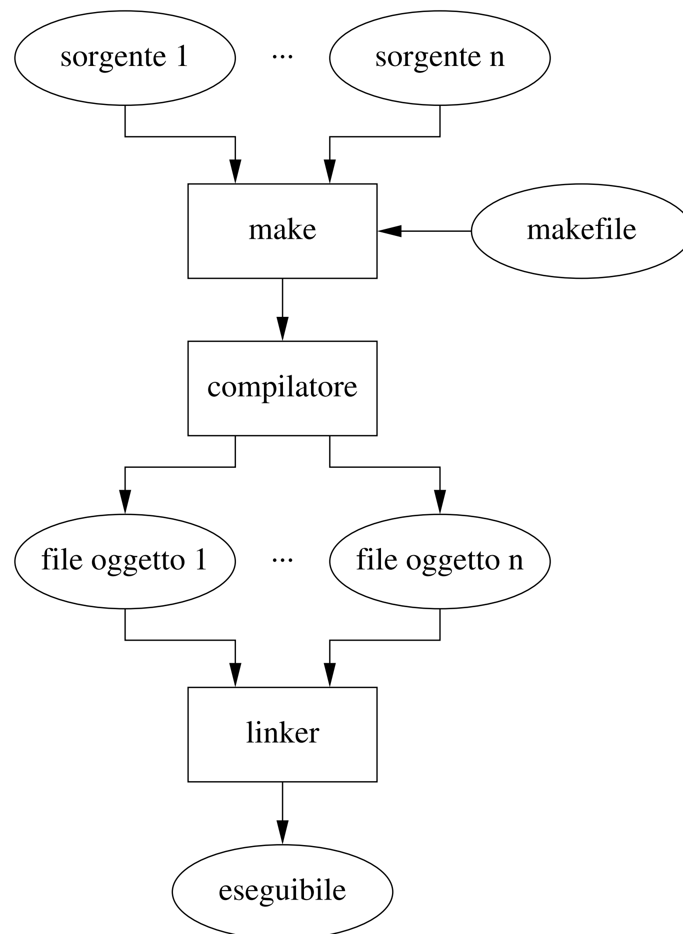

Figura 3.1: Schema descrittivo della realizzazione di un programma.

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 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 quanta informazione simbolica è presente 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.

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. salta al punto di inizio del programma

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 sia orientativamente quella di una schermata. Funzioni lunghe sono spesso suddivisibili in attività piu 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* di *istanza* tipiche di linguaggi più evoluti come Java o C++. È però possibile, come risulterà chiaro nel seguito, usare uno stile di programmazione orientata agli oggetti nella stesura dei propri programmi – ed è sempre meglio 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 e' il seguente[1].

```
/*
 * programma che stampa sul video la frase
 * Salve, mondo!
 */
#include <stdio.h>

int main()
{
```

[1]Il programma che stampa la stringa "Hello, world!" e' tradizionalmente riportato come primo programma di esempio quando si presenta un linguaggio di programmazione.

```
  printf("Salve, Mondo!\n");
  return 0;
}
```

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

- la presenza di un commento, che in questo caso illustra l'obiettivo del programma
- l'istruzione per includere un file di intestazione (`include`)
- la definizione della funzione `main`
- un'istruzione di stampa a video (`printf`)
- l'utilizzo di una stringa di testo[2] nella funzione `printf` delimitata dai doppi apici
- l'istruzione per ritornare il valore di uscita di una funzione (`return`)

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. Quando `main` termina, ovvero viene incontrata una istruzione `return` oppure si raggiunge l'ultima istruzione della funzione, il programma termina. Se ritorna zero vuol dire che il programma ha avuto successo, se ritorna non-zero vuol dire che c'è stato un errore; il numero specifico può indicare il tipo di errore riscontrato, se chi ha eseguito questo programma sa come differenziarli.

4.2 Compilazione del programma

Supponiamo di memorizzare il primo programma in un file chiamato `hello.c`. E' sempre bene denominare un file sorgente C dandogli un nome significativo, che rispecchi il contenuto del programma che vi e' implementato (ma questo e' una prassi generale da utilizzare per la denominazione di qualsiasi file!). In particolare, per chiarire che si tratta di un file sorgente in C, l'estensione del file sara' bene che sia `.c`.

Detto cio', il comando per la compilazione del programma e' il seguente:

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

Nell'istruzione precedente si possono distinguere le seguenti parti (in ordine logico diverso dal quale appaiono nel comando):

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

[2]Una stringa di testo e' sequenza di caratteri.

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

I messaggi del compilatore possono essere infatti di due tipi: segnalazioni di errore o warnings. In caso di errore, 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 e' 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 piu' insidiosi. Percio', come regola generale, e' bene realizzare programmi che generino quanti meno messaggi di avvertimento possibile.

Un esempio di esecuzione del programma di esempio e' la seguente:

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

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 delimitati da /* e */ oppure 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, non cosa fa, perché il cosa sta già nel codice. Questa regola vale per tutti i linguaggi, ma è così importante che val la pena di ripeterla.

4.4 Nota stilistica

Insieme ai commenti, uno degli aspetti che contribuisce di piu' alla realizzazione di programmi comprensibili e quindi piu' semplici da esaminare e manutenere anche a distanza di tempo, e' 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;}
```

e', dal punto di vista funzionale, esattamente identico al programma di esempio iniziale. Infatti e' una pura convenzione stilistica il fatto di scrivere le istruzioni su righe diverse, facendo opportunamente rientrare, ovvero *indentando*, le righe stesse. Questo perche' 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 composti da lettere, cifre (caratteri alfanumerici) e sottolineature "_" (*underscore*)
2. non possono essere delle parole chiave (vedi Sezione 4.6)
3. il primo carattere però non può essere numerico
4. il compilatore C distingue caratteri maiuscoli e minuscoli

NOTA L'uso di lettere maiuscole, per esempio in nomi come SortArrayOfNames, 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.

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.

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 e' 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 piu' importante relativo alle parole chiave e' 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 e' riservato per la definizione del linguaggio stesso.

Il motivo per cui le parole chiave non possono essere utilizzate come identificatori e' dato dal fatto che, se cio' 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 (gran parte di questi sono descritti nel Capitolo 7). Queste sono:

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

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.
Esempio:

```
/*
 * programma per il calcolo di una somma
 * stampa sul video del risultato
 */
#include <stdio.h>

int main()
{
  int a, b, c;

  a = 10;
  b = 12;
  c = a + b;
  printf("La loro somma e' %d\n", c);

  return 0;
}
```

Il programma dichiara le 3 variabili `a`, `b` e `c` di tipo `int` (intero), assegna un valore a `a` e `b`, calcola la loro somma, assegnandola alla variabile `c`, e 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, cio' che accade nel calcolatore e' 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`. Cio' che accade e' che i valori memorizzati nelle aree di memoria associate agli identificatori `a` e `b` vengono sommati all'interno del processore, dopo aver eventualmente trasferito (i valori potrebbero gia' trovarsi nei registri) tali valori dalla memoria centrale ai registri del processore. Il valore risultante e' poi scritto nell'area di memoria associata all'identificatore `c`.

Tenendo presente il significato dell'operatore =, e' 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 e' associabile a nessuna area di memoria.

Evidentemente il programma presentato non e' 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 piu' generale del programma potrebbe prevedere di leggere i dati da tastiera, come si vedra' nella Sezione 4.9, oppure in modo ancora piu' efficiente, leggendo i dati da un file.

4.8 Visualizzazione a video con `printf`

Nel semplice esempio precedente e' 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 della *variabile* `c` *di tipo* `int` sostituendo, all'interno della stringa `La loro somma e'`, la sequenza di caratteri che corrisponde al valore numerico della variabile. La funzione `printf` utilizza la sequenza di caratteri `%d` per individuare la posizione all'interno della stringa di partenza alla quale inserire il valore numerico. Nel caso specifico, `%d` informa la funzione `printf` che il tipo della variabile e' intero.

4.9 Lettura di dati da tastiera

Talvolta e' 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, che vengono assegnate ad opportune variabili[3].

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

```
/*
 * programma che legge i dati da tastiera,
 * ne effettua la somma e stampa sul video
 * del risultato
 */
#include <stdio.h>

int main()
{
  int a, b, c;

  printf("Scrivi due numeri interi\n");
  scanf("%d %d", &a, &b);
  c = a + b;
  printf("La loro somma e' %d\n", c);

  return 0;
}
```

Figura 4.1: Somma di due numeri letti da tastiera.

Il semplice programma in Figura 4.1 e' la versione un po' piu' raffinata del programma illustrato in Sezione 4.7, il quale legge due numeri da tastiera, li somma, e visualizza i risultati a video.

La chiave della lettura da tastiera risiede nella funzione `scanf`. Nell'esempio, la funzione `scanf` accetta due numeri interi immessi da tastiera e separati da uno spazio[4]. 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 e' 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 ne' stringhe di caratteri, ne' numeri a virgola mobile, ne' interi in notazioni diverse da quella decimale[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). Gli esempi presentati utilizzano la `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]L'input viene acquisito correttamente anche se viene premuto il tasto INVIO anche dopo aver inserito il primo numero.

[5]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`	intero in base 10; carattere di segno opzionale
i	`int`	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

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 e' sempre introdotta dal carattere %, seguito da una lettera che indica il tipo della variabile. Alcuni degli specificatori di formato piu' utilizzati sono riportati in Tabella 4.1, mentre la trattazione dettagliata dei vari tipi di dato e' riportata in Sezione 7.

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 alla fine della stringa.

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

Inoltre e' permesso specificare quante cifre devono essere stampate, per esempio decidendo di stampare un numero in virgola mobile con 2 sole cifre dopo la virgola.

Capitolo 5

Istruzioni e strutture di controllo

I COSTRUTTI DI CONTROLLO che verranno analizzati meglio nelle prossime sezioni sono i seguenti:

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

Una istruzione `istr` può essere un'espressione terminata da punto-e-virgola, un costrutto di controllo o un blocco delimitato da graffe. Il concetto di *espressione* include tutto, compresi gli assegnamenti a una variabile, tranne i costrutti di controllo.

Per la separazione di parole chiave, espressioni e ogni altro elemento atomico del linguaggi, le andate a capo, gli spazi e i tab sono equivalenti, e possono essere utilizzate per impaginare il codice sorgente del programma.

Lo stile di impaginazione è quindi libero, e programmatori diversi usano stili diversi. E' 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 ; }
```

e costituiscono un raggruppamento logico di istruzioni diverse. Per esempio[1]:

[1] Le istruzioni illustrate, e le istruzioni simili riportate successivamente, possono essere scritte in modo più conciso, come viene illustrato nel Capitolo A.

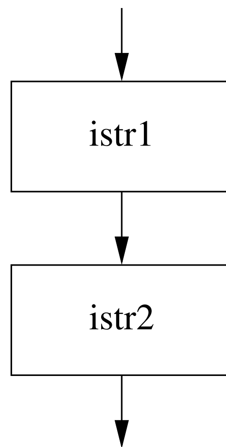

Figura 5.1: Istruzioni composte.

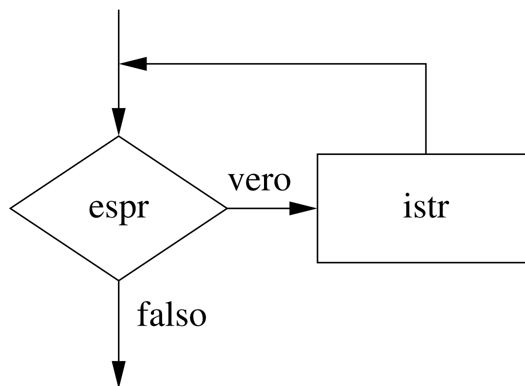

Figura 5.2: Il costrutto while.

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

L'operatore "," (virgola), del tipo

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

viene assegnato ad `a` il valore risultante dal calcolo di `espr2` dopo aver comunque calcolato `espr1`

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
```

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

```
#include <stdio.h>

int main()
{
  long i = 1, max;
  double somma = 0.0;

  printf("numero di iterazioni: ");
  scanf("%ld", &max);
  while (i <= max) {
    somma = somma + 1.0 / (i * i * i);
    i = i + 1;
  }
  printf("la somma è %f\n", somma);
  return 0;
}
```

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` minore o uguale 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 ) ;
```

Per esempio:

```
#include <stdio.h>

int main()
{
  long i = 1, max;
```

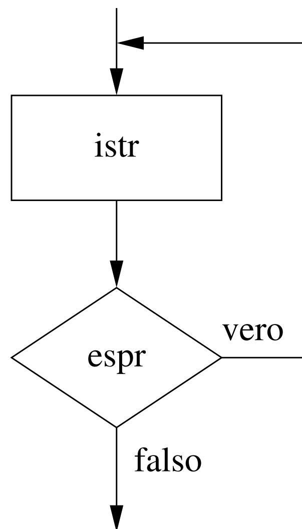

Figura 5.3: Il costrutto do-while.

```
  float somma = 0.0;

  printf("numero di iterazioni: ");
  scanf("%d", &max);

  do {
    somma = somma + 1.0 / (i * i * i);
    i = i + 1;
  } while (i < max);
  printf("La somma è %f\n", somma);
  return 0;
}
```

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
```

Un esempio è il seguente

```
#include <stdio.h>

int main()
{
```

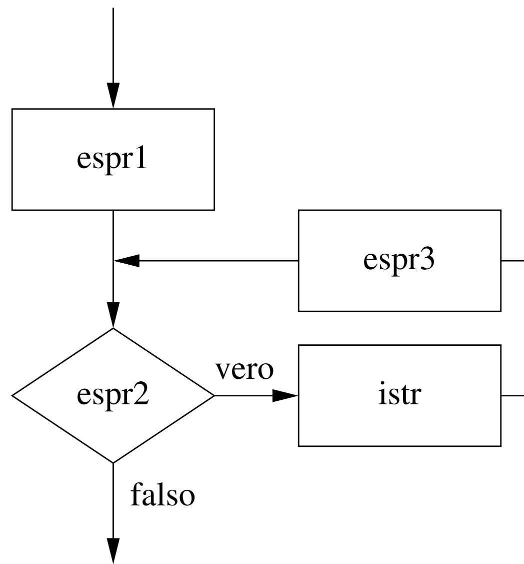

Figura 5.4: Il costrutto for.

```
  long i, max;
  float somma;

  printf("valore massimo di i: ");
  scanf("%ld", &max);

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

  printf("la somma è %f\n", somma);
  return 0;
}
```

La variabile `i`, che regola il numero di iterazioni, è solitamente chiamata *contatore*.

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 perchè, in questo caso, `MAX` indica direttamente il numero di iterazioni.

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

```
#include <stdio.h>

int main()
{
  long i, max;
  float somma;
```

```
  printf("numero di iterazioni: ");
  scanf("%d", &max);

  for(i = max, somma = 0; i > 0; i = i - 1)
    somma = somma + 1.0 / (i * i * i);

  printf("%f\n", somma);
  return 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 una variabile qualsiasi.

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

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 ;
```

```
#include <stdio.h>

int main()
{
  int a, b, max;

  scanf("%d %d", &a, &b);
  max = a;
  if (b > a)
    max = b;
  printf("il massimo è: %d", max);

  return 0;
}
```

Il costrutto `if` ammette l'istruzione 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

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

In Figura 5.7 il costrutto `if` è utilizzato per determinare il massimo tra due numeri interi.

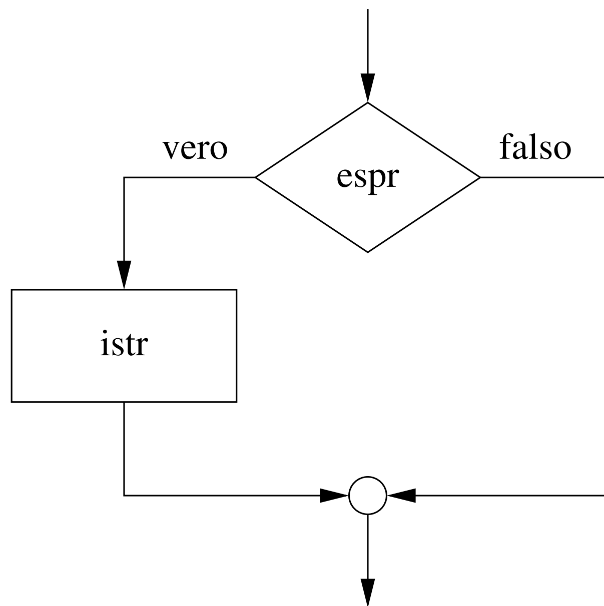

Figura 5.5: Il costrutto if.

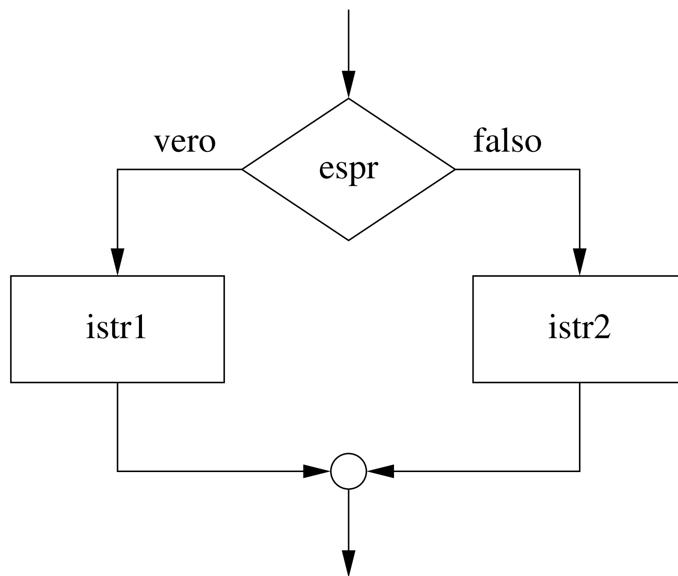

Figura 5.6: Il costrutto if-else.

```
#include <stdio.h>

int main()
{
  int a, b, max;

  scanf("%d %d", &a, &b);
  if (b > a) max = b;
  else max = a;
  printf("il massimo è: %d", max);

  return 0;
}
```

Figura 5.7: Costrutto `if` per la determinazione del massimo tra due numeri.

5.6 Il costrutto `switch`

Il costrutto di controllo `switch` serve a scegliere tra diversi comportamenti in base al valore di una espressione intera. La sintassi è diversa da quella degli altri costrutti di controllo, perchè 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 ; ]
}
```

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. Un carattere tra apici, cioè una costante di tipo `char`, è un numero intero.

La presenza di `break` alla fine di un caso è facoltativa, per permettere che le istruzioni associate ad un caso continuino con il codice del caso successivo; è sempre meglio commentare la mancanza di `break`, 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 sotto-ptogrammi che effettivamente eseguano le operazioni desiderate.

```
#include <stdio.h>

int main()
{
  char opt;

  printf("inserire il codice del comando (r, w, x): ");
  scanf("%c", &opt);

  switch(opt) {
    case 'r' :
    case 'R' :
      printf("lettura\n");
      break;
    case 'x' :
    case 'X' :
      printf("esecuzione\n");
      /* poi la scrittura */
    case 'w' :
    case 'W' :
      printf("scrittura\n");
      break;
    default:
      printf("operazione non valida\n");
      return -1;
  }
  return 0;
}
```

L'utente può inserire i caratteri `rwx` o `RWX`, che corrispondono alle seguenti operazioni:

r lettura

w scrittura

x esecuzione

Nel caso si richieda una esecuzione, le operazione 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, che portano all'esecuzione delle stesse istruzioni;
- l'uso dell'istruzione `break` per uscire dal costrutto `switch` nel caso di lettura;

- l'uso dell'istruzione `break` per uscire dal costrutto `switch` nel caso di sola scrittura;
- 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` 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, come è rara la necessità di scavalcare un `case` nell'esecuzione di quello precedente.

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 nella quale è contenuta;
- `continue` passa immediatamente al punto in cui viene effettuato il test del loop

Nell'esempio:

```
do {
  printf("valore di n: ");
  scanf("%d", &n);

  if (n < 0) {
    printf("  %d < 0: uscita dal loop...\n", n);
    break;
  }

  if (n > 10) {
    printf("  %d > 10: non ammesso\n", n);
    continue;
  }

  /* utilizza il valore di n */
  printf("  valore immesso: n = %d\n", n);

} while (1);
```

si ha un cosiddetto *ciclo infinito*: la condizione valutata nella `while` è sempre vera. Si ricorda 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`.

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.

E' 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`.

5.8 Il costrutto goto

L'istruzione goto esiste anche nel linguaggio C[2]. Il suo utilizzo è sconsigliato in quanto, in genere, complica la comprensione del programma, perchè rende difficile seguirne il flusso logico. Inoltre qualsiasi algoritmo può essere implementato usando i costrutti a disposizione senza l'utilizzo della goto, e quindi la disponibilità della goto è generalmente superflua.

Vi sono però delle eccezioni, per esempio quando si deve uscire da cicli più volte annidati a causa di situazioni anomale (errori). In pratica, la goto 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

```
while (1) {
  ....
  if (q) break;
  while (1) {
    ....
    /* esce solo dal ciclo più interno */
    if (p) break;
    ....
    /* esce dal ciclo più esterno */
    if (errore) goto uscita;
    ....
  }
}

uscita:
if (errore) { /* gestione del problema */  }
```

L'esempio sopra riportato consiste di due cicli annidati. 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.

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 al punto indicato con l'etichetta uscita.

[2] E' un costrutto utilizzato moltissimo in altri linguaggi, come il BASIC.

Capitolo 6

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 A.

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.

E' 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 illustati tutti gli operatori disponibili nel linguaggio C, nello stesso ordine di precedenza indicato in Appendice A.

6.1 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
```

```
(espr1 op1 espr2) op2 espr3
espr1 op1 (espr2 op2 espr3)
```

La prima espressione è composta da più operatori senza parentesi che ne cambiano la precedenza. Se `op1` ha precedenza maggiore di `op2`, viene prima valutata l'espressione `espr1 op1 espr2`, ed il risultato viene composto a destra con `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 `espr3`.

6.2 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

6.3 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.

6.4 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.

6.5 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` *p equivale a vet[0].

L'espressione `*++p = 10;` (vet[0] = 10; *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. Da notare che l'espressione `(*p)++` non può essere invece scritta se non con l'uso delle parentesi. In questo caso, verrebbe incrementato il contenuto (*p) a cui punta p, ma dopo del suo utilizzo a causa dell'operatore di incremento ++ usato in forma postfissa.

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.

6.6 Chiamata a funzione

operatore	( ) parentesi tonde
sintassi	`nome_funzione ( parametri )`
n. operandi	1
utilizzo	chiamata della funzione `nome_funzione`
associativita'	$\Longrightarrow$
commutativita'	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

```
void *p = malloc(1024);
```

le parentesi tonde indicano la chiamata della funzione di libreria `malloc`, che delimitano l'elenco dei parametri che in questo caso è uno solo, 1024.

```
extern int (*rd_data)(void *buffer, int count);
int result = rd_data(p, 1024);
```

6.7 Elemento di vettore

operatore	[ ] parentesi quadre
sintassi	`vett [ id ]`
n. operandi	2
utilizzo	accesso all'elemento di indice `id` del vettore `vett`
associativita'	$\Longrightarrow$
commutativita'	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;
```

6.8 Elemento di struttura

operatore	. punto
sintassi	`strutt . camp`
n. operandi	2
utilizzo	accesso al campo `camp` della struttura `strutt`
associativita'	$\Longrightarrow$
commutativita'	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, `st_mode` è un campo di tipo intero della struttura `struct stat` dichiarata nel file di intestazione `stat.h`.

```
#include <sys/stat.h>
/* st_mode è un campo intero di struct stat */

struct stat st, *stptr, stvec[10];
int i;

i = st.st_mode;
i = (*stptr).st_mode;
i = stvec[5].st_mode;
```

Si noti che il puntatore `stptr` deve essere allocato o assegnato prima di utilizzarlo.

Le tre istruzioni di assegnamento dell'esempio non sono equivalenti, in quanto possono assegnare alla variabile `i` dei valori diversi. Si noti l'utilizzo dell'operatore "." che indica il campo specifico della struttura `struct stat`.

6.9 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`
associativita'	$\Longrightarrow$
commutativita'	NO

E' 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.

```
#include <sys/stat.h>

/* st_mode è un campo intero di struct stat */
struct stat st, *stptr, stvec[10];
int i;

/* funzione ipotetica */
extern struct stat *getstatptr(int i);

i = stptr->st_mode;
i = (stvec+5)->st_mode;
/* & tra parentesi, -> ha priorità maggiore */
i = (&st)->st_mode;
getstatptr(5)->st_mode;
```

6.10 Negazione logica

operatore	!
	punto esclamativo
sintassi	! `espr`
n. operandi	1
utilizzo	negazione logica dell'espressione `espr`
associativita'	$\Longleftarrow$
commutativita'	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.

```
p = malloc(128);

if (!p) {
  /* gestione errore */
}

/* ritorna 0 se i vale 0, 1 se i e' diverso da zero */
return !!i;
```

6.11 Complemento a 1

operatore	$\sim$
	tilde
sintassi	$\sim$ `val`
n. operandi	1
utilizzo	complemento a 1 – negazione dei singoli bit di `val`
associativita'	$\Longleftarrow$
commutativita'	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.

```
/* 0xffffffff se 32 bit, 0xff se 8 bit, eccetera */
i = ~0;

/* 0xfffff000 se la pagina e' 4k cioe' 0x1000*/
int page_mask = ~(PAGE_SIZE-1)
```

6.12 Negazione unaria

operatore	$-$
	trattino alto – segno meno
sintassi	$-$ `espr`
n. operandi	1
utilizzo	cambia segno all'espressione `espr`
associativita'	$\Longleftarrow$
commutativita'	NO

La negazione unaria nega l'espressione aritmetica alla sua destra, cioè cambia di segno al risultato dell'espressione.

```
i = -1;

/* EINVAL è un codice di errore intero positivo */
return -EINVAL;
```

6.13 Incremento e decremento

operatore	++ più più
sintassi	`val++` oppure `++val`
n. operandi	1
utilizzo	incrementa di una unità il valore di `val`
associativita'	$\Longleftarrow$
commutativita'	NO

operatore	−− meno meno
sintassi	`val--` oppure `--val`
n. operandi	1
utilizzo	decrementa di una unità il valore di `val`
associativita'	$\Longleftarrow$
commutativita'	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 6.30 sull'operatore di assegnamento).

```
/* stack pointer che punta al primo elemento vuoto */
int stack[10], sp = 0;

/* inserimento ("push") */
stack[sp++] = datum;

/* estrazione ("pop") */
datum = stack[--sp];

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 */
}
```

6.14 Estrazione di un indirizzo

operatore	&
	"e" commerciale
sintassi	`& espr`
n. operandi	ritorna l'indirizzo di `espr`
utilizzo	1
associativita'	$\Longleftarrow$
commutativita'	NO

Esempio:

```
#include <sys/stat.h>
struct stat stbuf;

/* la funzione chiamata scrive in stbuf */
stat("/bin/sh", &stbuf);

char s[32];

/* sscanf converte da stringa e scrive in i */
sscanf(s, "%i", &i);
```

6.15 Uso di puntatore

operatore	*
	asterisco
sintassi	`* ptr`
n. operandi	dereferenzia il puntatore `ptr`
utilizzo	1
associativita'	$\Longleftarrow$
commutativita'	NO

L'operatore permette l'uso del valore contenuto all'indirizzo memorizzato in un puntatore.

```
int v[32], *p;
for (p = v + 32; p >= v; p--)
  sum = sum + *p;
```

6.16 Operatore di casting

Il *casting* di una variabile significa cambiare il tipo di dato assiciato alla variabile stessa.

operatore	( ) parentesi tonde
sintassi	`( type ) espr`
n. operandi	2
utilizzo	cambia il tipo di `espr` in `type`
associativita'	$\Longleftarrow$
commutativita'	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 */ }
```

6.17 Dimensione di una variabile

operatore	`sizeof` “`sizeof`”
sintassi	`sizeof var`
n. operandi	1
utilizzo	ritorna la dimensione in byte di `var`
associativita'	$\Longleftarrow$
commutativita'	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.

```
struct buf *buffers = malloc(10 * sizeof(struct buf));

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 */
}
```

6.18 Moltiplicazione e divisione

operatore	*
	asterisco
sintassi	`espr1 * espr2`
n. operandi	2
utilizzo	moltiplica `espr1` per `espr2`
associativita'	$\Longrightarrow$
commutativita'	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.

operatore	/
	slash
sintassi	`espr1 / espr2`
n. operandi	2
utilizzo	divide `espr1` per `espr2`
associativita'	$\Longrightarrow$
commutativita'	NO

La divisione intera scarta il resto.

```
/* conversione di temperatura */
fahr = cels * 9 / 5;

/* errato per perdita del resto: usare "* 5 / 9" */
cels = fahr / 9 * 5;

/* overflow se sec < -2 o sec > 2 */
int nsec = sec * 1000000000
```

6.19 Resto di divisione intera

operatore	%
	percentuale
sintassi	`espr1 % espr2`
n. operandi	2
utilizzo	restituisce il resto della divisione intera tra `espr1` e `espr2`
associativita'	$\Longrightarrow$
commutativita'	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);
}
```

6.20 Somma e sottrazione

operatore	+
	più
sintassi	`espr1` + `espr2`
n. operandi	2
utilizzo	restituisce la somma tra `espr1` e `espr2`
associativita'	$\Longrightarrow$
commutativita'	SI'

operatore	−
	trattino alto
sintassi	`espr1` − `espr2`
n. operandi	2
utilizzo	restituisce la differenza tra `espr1` e `espr2`
associativita'	$\Longrightarrow$
commutativita'	NO

Se uno dei due operandi può essere un puntatore, in tal caso il risultato è un puntatore. 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 */
```

6.21 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`
associativita'	$\Longrightarrow$
commutativita'	NO

operatore	>> maggiore maggiore
sintassi	`var` >> `n`
n. operandi	2
utilizzo	effettua lo scorrimento a destra di `n` posizioni dei bit di `var`
associativita'	$\Longrightarrow$
commutativita'	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 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 da 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);
```

La codifica RGB[1] utilizza un certo numero di bit per memorizzare le rispettive componenti di colore. 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 . Big-endian e little-endian sono due metodi alternativi per memorizzare valori numerici che richiedono più di 8 bit. 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 0x12345678 (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.

[1] Red–Green–Blue, ovvero Rosso–Verde–Blu.

6.22 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
associativita'	$\Longrightarrow$
commutativita'	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
associativita'	$\Longrightarrow$
commutativita'	NO

Esempio

```
int fuorimisura = i > 100 || i < 50;

/* conversione da cm in decimi di pollice */
i = i * 1000 / 254;
if (fuorimisura) {
  /* gestione errore */
}
```

6.23 Confronto: uguaglianza e diversità

operatore	==
	uguale uguale
sintassi	`espr1 == espr2`
n. operandi	2
utilizzo	ritorna 1 se `espr1` è uguale a `espr2`, altrimenti 0
associativita'	$\Longrightarrow$
commutativita'	SI'

operatore	! =
	punto esclamativo – uguale
sintassi	`espr1` ! = `espr2`
n. operandi	2
utilizzo	ritorna 1 se `espr1` è diverso `espr2`, altrimenti 0
associativita'	$\Longrightarrow$
commutativita'	SI'

Gli operatori di confronto servono a verificare uguanglianza 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.

6.24 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`
associativita'	$\Longrightarrow$
commutativita'	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.

6.25 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`
associativita'	$\Longrightarrow$
commutativita'	SI'

Il risultato dello XOR tra due valori binari vale 1 se entrambi i valori sono diversi.

```
while ( /* condizione */ ) {
   /* calcolo */
   led = led ^ 1; /* inversione del bit più basso */
}
```

Se `led` vale 11010001, dopo l'operazione diventa 11010000 e viceversa.

6.26 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`
associativita'	$\Longrightarrow$
commutativita'	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 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 */
```

6.27 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
associativita'	$\Longrightarrow$
commutativita'	SI'

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.

Per esempio[2]:

```
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:

- controlla che `strptr` non sia nullo;
- controlla che `methods` non sia nullo;
- controlla che `print` non sia nullo.

L'AND vale 1 solo se TUTTI i puntatori sono diversi da 0.

6.28 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
associativita'	$\Longrightarrow$
commutativita'	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[3]:

```
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 perchè 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

[2]Vedi programma di esempio `and.c`.
[3]Vedi programma di esempio `or.c`.

1. prima si controlla che il puntatore non sia già non-nullo, perchè 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.

6.29 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`
associativita'	⟸
commutativita'	NO

E' 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;
```

6.30 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`
associativita'	$\Longrightarrow$
commutativita'	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.

```
/* 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 è assegnabile */
"nome" = s;

/* lvalue non valida: come sopra ma più evidente */
3 = i;
```

Nell'esempio

```
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.

6.31 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`
associativita'	$\Longleftarrow$
commutativita'	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;
```

6.32 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"
associativita'	$\Longrightarrow$
commutativita'	NO

E' usato principalmente nei cicli `while` e per fare cicli `for` con due o più indici.

```
while(next_number(&i), i) {
  /* il nuovo i è diverso da zero */
}

for (p = v, i = 0; i<32; p++, i++) {
  /* p scorre il vettore fino a
   * un massimo di 32 elementi
   */
}
```

Capitolo 7

Tipi di dati

In un linguaggio di programmazione, un tipo di dato è un nome che identifica l'insieme di valori che possono essere assunti da una variabile o da una espressione.

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, in modo da informare esplicitamente il compilatore sul tipo (appunto...) di 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 numeri interi, in virgola mobile, o puntatori. 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.

7.1 Tipi interi

I tipi interi predefiniti del linguaggio sono riportati in Tabella 7.1.

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 viene usato un bit 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

Tabella 7.1: 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

Tabella 7.2: 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

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 7.2.

Sulla lunghezza (dimensione in bit) di tali tipi non si possono fare assunzioni, poichè 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 di 8 bit.

La definizione 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, 64 bit in base alla macchina in uso[1]. Viene comunque garantito che una variabile di tipo `int` possa utilizzare almeno 16 bit.

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 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 una codifica ASCII una costante di tipo `char` che vale '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 perchè il carattere '1' equivale al valore numerico 49 decimale.

In Tabella 7.1 dei caratteri ASCII stampabili completa è riportata di seguito:

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

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

Figura 7.1: Tabella ASCII.

7.1.1 Conversione da esadecimale a decimale

Per illustrare la dualità tra il tipo `char`, il valore intero che esso rappresentano, e l'associazione con i caratteri ASCII, viene proposto il seguente spezzone di codice, che implementa la conversione da esadecimale a decimale. La conversione proposta è estremamente inefficiente, in quanto i caratteri vengono convertiti uno alla volta.

```
int value;

int nextchar(int c)
{
  switch(c) {
    case 'a': case 'b': case 'c': case 'd: case 'e': case 'f':
      c = c - 'a' + 10 + '0';
    /* fall through */
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
      value = value * 10 + c - '0';
      break;
    case 'p':
      printf("%i\n", value); value = 0;
      break;
    default:
      return -1; /* error */
  }
  return 0;
}
```

Nell'algoritmo presentato, l'istruzione

```
c = c - 'a' + 10 + '0';
```

è necessaria in quanto i caratteri ASCII nel range $['a' \ldots' f']$ non sono consecutivi ai caratteri $['0' \ldots' 9']$. Nel caso il valore *del carattere* `c` in ingresso sia nel range $['a' \ldots' f']$ delle cifre esadecimali, assegna a `c` il valore dei caratteri seguenti al range $['0' \ldots' 9']$, in modo che la successiva istruzione

```
value = value * 10 + c - '0';
```

possa correttamente calcolare il valore di `value` moltiplicando per 10 il valore corrente, e sommandovi il valore intero di `c` dal quale è sottratto lo scostamento della prima cifra ASCII ('0') dell'intervallo.

L'istruzione precedente viene eseguita sia nel caso che il valore di `c` in ingresso sia una lettera che nel caso sia una cifra, infatti non c'è il `break` dopo il primo insieme di clausole `case`.

il valore di `c` viene modificato dopo essere stato usato per la selezione del caso corretto; non deve stupire, in quanto l'espressione di `switch` viene valutata una volta sola.

7.2 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:

Tabella 7.3: I tipi a virgola mobile previsti nel linguaggio C.

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

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

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

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

A seconda del numero di bit dedicati a ciascun campo, si possono distinguere tipi floating point caratterizzati da diversa precisione. I tipi a virgola mobile disponibili in C sono elencati in Tabella 7.3

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.

7.3 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

```
int *p;
```

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

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 pò ostico del linguaggio C, verranno ripresi in modo più dettagliato al Capitolo 8.

7.4 I vettori

I vettori, o array, 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 [ cardinalità ] ;
```

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 num[10];
```

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

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

Il programma seguente effettua la somma di due numeri che, invece di essere memorizzati in variabili singole, vengono memorizzati come elementi di un vettore opportunamente dimensionati.

```
/*
 * programma per il calcolo di una somma
 * utilizzando dati memorizzati in un vettore
 */
#include <stdio.h>
```

```
int main()
{
  int v[3];

  v[0] = 10;
  v[1] = 12;
  v[2] = v[0] + v[1];
  printf("La loro somma è %d\n", v[2]);

  return 0;
}
```

Nel caso in cui il vettore venga indicizzato con un indice al di fuori del range ammesso, nel migliore dei casi si ha un errore di "segmentation fault", mentre in altri casi si va ad scrivere/-leggere una variabile che appartiene al programma, allocata subito dopo al 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.

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.

E' possibile dichiarare anche vettori multi-dimensionali. Il caso tipico è quello di vettori a fue dimensioni, ovvero le cosiddette matrici. Un esempio di dichiarazione di matrici è la seguente:

```
nome-tipo identificatore [ cardinalità ] [ cardinalità ] ;
```

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 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]`.

7.5 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:

```
int count;
struct stat stbuf;
struct stat *stptr;
```

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. In questo esempio le tre strutture sono uguali, con il campo priv inizializzato a zero:

```
struct item {
  int id;
  char *name;
  int value;
  int priv;
};

struct item i1 = {3, "aldo", 45};
struct item i2 = {id: 3, name: "aldo", value: 45};
struct item i3 = {.id = 3, .name = "aldo", .value = 45};
```

Un altro esempio di strutture dati è la seguente

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

  struct data giorno3;
```

Ho definito un nuovo tipo di nome `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>
```

per esempio:

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

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

è stata definita una variabile di nome `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`.

7.5.1 Altri esempi di uso delle strutture

Nelle applicazioni matematiche può essere utile definire una struttura per i numeri complessi[2], per esempio come segue:

```
struct complex {
  float real;
  float imm;
};
```

e quindi delle funzioni che operano su numeri complessi, che abbiano come parametri strutture, ma soprattutto possano ritornare come valore una struttura:

```
struct complex per(struct complex z1, struct complex z2)
{
  struct complex ris;

  ris.real = z1.real * z2.real -
    z1.imm * z2.imm;
  ris.imm = z1.real * z2.imm +
    z1.imm * z2.real;

  return ris;
}

struct complex piu(struct complex z1, struct complex z2)
{
  struct complex ris;

  ris.real = z1.real + z2.real;
  ris.imm = z1.imm + z2.imm;

  return ris;
}
```

e quindi utilizzare, per esempio come segue:

[2]L'uso di numeri complessi è talmente diffuso in applicazioni matematiche che nell'ultima revisione del linguaggio C, il C99, il tipo complesso è stato introdotto come tipo nativo.

```
struct complex z1, z2, z3, z4, ris;

ris = piu(per(z1, z2), per(z3, z4));
```

è l'implementazione dell'espressione complessa

$$ris = z_1 z_2 + z_3 z_4$$

7.6 Union

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

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

A differenza delle strutture con le `union` si riserva spazio in memoria sufficiente solo per il dato più grande fra i vari campi descritti.

Esempio:

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

su alcuni sistemi viene stampato 3031 su altri 3130.

ATTENZIONE: generalmente un codice che contiene `union` non è portabile.

7.7 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`).

7.8 Conversioni di tipo

Le conversioni di tipo, o *casting*, permettono di trasformare un valore contenuto in una variabile di un determinato tipo in un valore opportuno di un tipo diverso.

La conversione può avvenire in vari modi:

Conversioni automatiche:

- in espressioni che coinvolgono tipi diversi, il risultato dipende dall'operando più ricco di informazioni
- es. 8/5 coinvolge due interi, il risultato è intero 1
- es. 8/5.0 coinvolge un intero ed un double, il risultato è double 1.6

Conversioni per assegnamento:

- il valore della parte destra viene convertito nel tipo della parte sinistra
- `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)
- `int` → `char` vengono conservati i bit meno significativi
- es. `char c`; `c = 257`; `c` vale 1

Conversioni esplicite (operatore cast):

- (nome tipo) espressione
- es. `8/(double) 5` risultato 1.6 (5 viene forzato a `double`) ma
- es. `(double) (8/5)` risultato 1.0 (prima si calcola la divisione fra interi, poi il risultato viene considerato `double`)

7.9 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. Un'altra possibile ragione è quella di realizzare 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. Ma un qualsiasi programma applicativo userà sempre il nome `int16`.

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

```
typedef tipo nome;
```

che associa il nome `nome` al tipo `tipo`.

Per esempio, in un sistema di calcolo che tiene traccia del trascorrere del tempo in unità discrete, è possibile effettuare il seguente assegnamento:

```
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 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.

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

Capitolo 8

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.* E' possibile, per esempio, memorizzare l'indirizzo di un'altra variabile del programma.

In Figura 8.1 è rappresentata la variabile `i` di dimensione 4 bytes, il cui indirizzo è pari a 100. Un puntatore che punta alla variabile `i` conterrà quindi il valore 100. La situazione è riassunta in Tabella 8.1. Si noti che l'indirizzo *a cui è allocato* il puntatore `pt` non è rilevante, 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 *pt;
```

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

L'operatore unario & permette di ottenere l'indirizzo di un oggetto (in genere una variabile)

```
pt = &i;
```

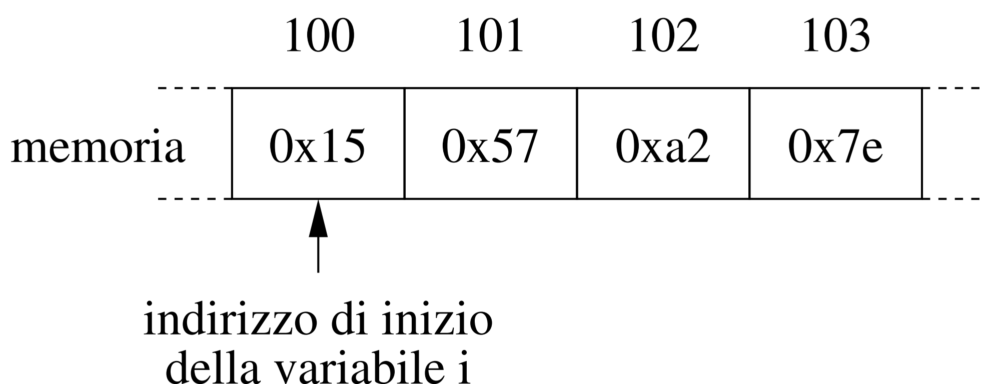

Figura 8.1: Una variabile in memoria.

Tabella 8.1: Lo stato rappresentato in Figura 8.1..

variabile	indirizzo	valore
i	100	0x1557a27e
pt	non rilevante	100

in `pt` è stato memorizzato l'indirizzo della variabile `i`.

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

```
i = *pt;
```

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

8.1 Puntatori e vettori

Per definizione, il valore costante di una variabile di tipo vettore è l'indirizzo 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.

Nel seguente esempio:

```
#include <stdio.h>

int main()
{
  int vett[10];
  int *p;
  int i;

  p = vett;

  for (i = 0; i < 10; i++)
    *(p + i) = 100 + i;

  for (i = 0; i < 10; i++)
    printf("%d ", vett[i]);
  printf("\n");

  return 0;
}
```

viene dichiarato il vettore `vett` di 10 elementi interi, e il puntatore ad intero `p`. L'istruzione

```
  p = vett;
```

assegna al puntatore `p` l'indirizzo del primo elemento del vettore `vett`. Il vettore viene riempito con il primo ciclo, 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 accedendo al vettore attraverso il suo identificatore `vett`.

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.

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 di puntatore oggetto di 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 è un numero intero). L'aritmetica su puntatori generici (puntatori a void) con gcc usa 1 byte come dimensione dell'elemento puntato, mentre non è definita secondo lo standard del linguaggio. Il "puntatore nullo" vale zero e non è un puntatore valido; nelle funzioni che ritornano un puntatore è spesso ritornato come segnalazione di errore. La macro NULL vale 0, e 0 è confrontabile con qualunque puntatore.

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:
                    "è intero ogni elemento del vettore v, di dimensione 10"
                    "è intero quello che è puntato da p" */
p = v;          /* assegno a p il valore di v, 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 p */
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. 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 linguaggio 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;
```

È un errore fare assegnamenti tra puntatori di tipo diverso, così come è un errore fare operazioni aritmetiche tra puntatori di tipo diverso. 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 fatte le seguenti dichiarazioni e istruzioni:

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

p = &vet[0];
```

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;
```

8.2 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 incremento un puntatore `p`, il suo valore numerico (indirizzo in memoria in byte) viene incrementato di `sizeof(*p)`.

Esempio:

```
int i, v[10], *p;      /* le stesse variabili di prima */
i = sizeof(int);       /* normalmente 4, ma può 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 */
```

8.3 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:

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

è cioè un vettore di caratteri in grado di memorizzare al più 9 caratteri (l'ultimo è sempre 0).

```
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 rappresntare 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 apice singolo 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 è "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 è un indirizzo costante */
surname++;
```

Una implementazione di `strlen`, funzione che ritorna la lunghezza di una stringa, può quindi essere la seguente:

```
int strlen(char *s)
{
  char *t = s;
  for (; *t; t++)
        ;
  return t - s;
}
```

8.4 Puntatori e strutture

Una struttura dati può includerne altre o includere puntatori ad altre strutture. 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 father;

struct child {
  struct father *father;
  /* ... */
};

struct father {
  struct child *child;
  /* ... */
};
```

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 dpriv;
struct datum {
  struct dpriv *priv; /* lista dei campi ignota all'utente di datum */
  /* ... */
  struct datum *next; /* per l'inserimento in una lista */
};
```

8.5 Argomenti del programma

La funzione `main` ha due parametri che convenzionalmente si chiamano `argc` e `argv` definiti rispettivamente come `int` e `char **`.

```
int main(int argc, char ** argv);
```

oppure

```
int main(int argc, char * argv[]);
```

La prima variabile memorizza il numero di argomenti sulla riga di comando, la seconda memorizza l'indirizzo del primo elemento di un vettore di puntatori a carattere. Ogni puntatore fa riferimento al primo carattere di ogni argomento. Supponiamo di avere il programma `saluta` e di richiamarlo con la riga di comando:

```
$ saluta Tizio, Caio
```

In questo caso `argc` vale 3, e

- ad `argv[0]` è associata la stringa "saluta"
- ad `argv[1]` "Tizio," (notare la presenza della virgola)
- ad `argv[2]` "Caio"

8.6 Puntatori a funzione

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. 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. 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 a altre funzioni, per esempio la funzione di libreria `qsort`, funzione che implementa l'algoritmo "quick sort" su un vettore. Il compilatore verifica in compilazione che i tipi dei puntatori a funzione siano compatibili, cioè le funzioni come dichiarate ricevano gli stessi argomenti. Esempio:

```
#include <string.h> /* per la dichiarazione di strcmp */
#include <stdlib.h> /* per la dichiarazione di qsort */

char *strings[100]; /* definisco un vettore di 100 puntatori */

strcmp(strings[0], strings[1]); /* confronto due stringhe */

/* chiamo qsort dicendo che strcmp() è la funzione di confronto da usare */
qsort(strings, 100, sizeof(char *), strcmp);

strncmp(strings[0], strings[1], 5); /* confronto solo i primi 5 caratteri */

/* questo invece è un errore, perché strncmp riceve tre argomenti */
qsort(strings, 100, sizeof(char *), strncmp);
```

Capitolo 9

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.

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 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[1]. Tranne, ovviamente, per le funzioni variadiche (Sezione 9.6), nel qual caso possono essere passati un numero arbitrario argomenti ulteriori, anche 0.

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.

9.1 Dichiarazione di funzioni

In una dichiarazione (*prototyping*) si riconoscono:

- il tipo di dato restituito dalla funzione, se non specificato tipo `int`, `void` se la funzione non restituisce alcun valore
- il nome della funzione
- i tipi degli eventuali argomenti, `void` o niente se la funzione non usa parametri
- termina con ';'

[1] Non esiste il polimorfismo delle funzioni in C, concetto fondamentale nei linguaggi di programmazione ad oggetti come C++ o Java.

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.

Esempio:

```
void proc1(int arg1, double arg2);
```

definisce una funzione che non restituisce alcun valore con due argomenti: un intero ed un reale in doppia precisione.

Altro esempio:

```
char proc2(void);
```

definisce una funzione che restituisce un carattere.

9.2 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
   - istruzioni
   - una o più istruzioni `return`

Una funzione come

```
do_nothing(){ }
```

è lecita.

9.3 Passaggio dei parametri

Una funzione può utilizzare parametri per svolgere il suo lavoro, cioè delle variabili che vengono passate 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*.

```
#include <stdio.h>

/* dichiarazione di funzione */
float massimo(float, float);

int main()
{
  int a, b;

  scanf("%d %d", &a, &b);
  printf("il massimo è: %f",
    /* invocazione di funzione */
    massimo(a,b)
  );
  return 0;
}

/* definizione di funzione */
float massimo(float a, float b)
{
  return b > a ? b : a;
}
```

Figura 9.1: Esempio di passaggio dei parametri.

Un esempio di passaggio dei parametri ad una funzione creata dall'utente è riportato in Figura 9.1.

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.

9.4 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.

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.

Perchè 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. Esempio:

```
int findid(struct obj *item, int *errorptr)
{
  if (isvalid(item) == 0) {
    *errorptr = ERR_INVALID;
    return 0;
  }
  *errprptr = ERR_NOERROR;
  return internal_findid(item);
}
```

Nell'esempio precedente, la funzione `findid` viene utilizzata per trovare un valore intero all'interno della struttura passata come argomento (anche la struttura è passata per riferimento, per i motivi di efficienza appena illustrati). Quindi il tipo restituito dalla funzione `findid` è intero. Per ricercare il valore, `findid` si appoggia alla funzione `internal_findid`, che accetta il puntatore alla struttura come parametro. Prima di chiamare `internal_findid`, `findid` effettua dei controlli di validità sulla struttura, in quanto `internal_findid` si aspetta in ingresso una struttura corretta. Inoltre, `internal_findid` deve notificare al chiamante il fatto di aver ricevuto o meno una struttura corretta, in modo che il chiamante non interpreti erroneamente il valore di ritorno di `findid`. Ecco che il secondo parametro di `findid` viene modificato internamente alla funzione per assegnargli il codice di errore e, di fatto, ritornare un secondo valore di uscita. Un possibile uso della funzione dell'esempio è il seguente:

```
struct obj item;
int id, errcode;

id = findid(&item, &errcode);
if (errcode == ERR_INVALID) {
  /* gestisce l'errore, per esempio: */
  printf("Struttura non valida. Code di errore: \n", errcode);
  exit(1);
}
/* utilizza il valore restituito */
```

9.5 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.

```
#include <stdio.h>

int main(int argc, char ** argv)
{
  int i;

  for (i = 0; i < argc; i++) {
    printf("Argomento %d (%d): %s\n", i, argc, argv[i]);
  }

  return 0;
}
```

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.

9.6 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.

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, perchè ad un nome può essere associato un solo indirizzo, sia esso codice o dati.

10.1 La visibilità (scope) 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*.

Le variabili possono anche essere dichiarate al di fuori del corpo delle funzioni o di un 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.

```
/*
 * scope.c
 * Esempi di visibilità delle variabili
 */
#include <stdio.h>

int c = 1;

void func1(int c)
{
  printf("[func1, pre if ] %d\n", c);
  if (1) {
    int c = 5;
```

```
    printf("[func1, if     ] %d\n", c);
  }
  printf("[func1, post if] %d\n", c);
}

void func2()
{
  printf("[func2, pre if ] %d\n", c);
  if (1) {
    int c = 6;
    printf("[func2, if     ] %d\n", c);
  }
  printf("[func2, post if] %d\n", c);
}

int main(int argc, char **argv)
{
  int c = 2;

  printf("[main,  pre if ] %d\n", c);
  if (1) {
    int c = 3;
    printf("[main,  if     ] %d\n", c);
  }
  printf("[main,  post if] %d\n", c);

  func1(4);
  func2();

  return 0;
}
```

La variabile `c` è definita globalmente, 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 all'esterno del blocco suo `if`, invece della variabile globale, mentre la funzione `func2` utilizza, sempre all'esterno del suo blocco `if`, la variabile globale. La funzione `main`, invece, dichiara localmente una variabile chiamata `c`, che viene quindi utilizzata al posto della variabile globale.

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*. 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.

Tabella 10.1: Scope e lifetime delle variabili.

Classe	Scope	Lifetime
extern	globale	permanente
auto	locale	temporaneo
static	locale	permanente
register	locale	temporaneo

Il tempo di vita è permanente per le variabili allocate staticamente, mentre è temporaneo per le variabili allocate dinamicamente, che vengono rilasciate al termine dell'esecuzione del blocco in cui sono definite.

```
#include <stdio.h>

int c1 = 1;

void func() {
  int c3 = 3;
  static c4 = 1;

  printf("func: c1 %d c3 %d c4 %d\n", c1, c3, c4);
  c4++;
}

int main(int argc, char **argv)
{
  int i;

  printf("main: c1 %d\n", c1);

  for (i = 0; i < 5; i++) func();

  return 0;
}
```

La variabile `c1` è globale, allocata staticamente, e può essere utilizzata in tutto il file sorgente. La variabile `c3` è allocata dinamicamente ogni volta che la funzione `func` viene richiamata e può essere utilizzata soltanto all'interno della funzione stessa. Per quanto riguarda la variabile `c4`, anche se è dichiarata all'interno della funzione `func`; 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.

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[1] e nemmeno la raccolta della spazzatura (*garbage collection* sembra più raffinato, ma di quello si tratta).

La memoria usata dai programmi può essere di tre tipi: statica, dinamica, automatica. Una variabile o struttura dati statica è quella dichiarata in compilazione, cui il linker assegna un indirizzo immutabile. Una struttura dati dinamica è allocata durante il funzionamento del programma, per esempio chiamando `malloc` e accedendo allo spazio così ottenuto tramite un puntatore. Una variabile cosiddetta "automatica" 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 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 memoria che viene richiesta al sistema durante il funzionamento del programma. Al momento dell'allocazione non si possono fare assunzioni sul 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 la dimensione del programma in esecuzione aumenterà in continuazione. Mentre la memoria di un programma in spazio utente viene liberata tutta al termine del programma, una allocazione non liberata in spazio kernel causa una perdita di memoria che può essere recuperata solo riavviando la macchina.

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.

Esempi:

```
int i;              /* dato inizializzato a zero, segmento bss */
int v[4] = {2,1,};  /* dato inizializzato a {2,1,0,0}, segmento dati */
int j = f(3, i);    /* errore: valore non noto in compilazione */

int *f(int x, int y)
{
  int z;                           /* var. automatica, non so quanto vale */
  int a=0, b=1, c=2;               /* variabili  inizializzate a run-time */
  int *p = malloc(4 * sizeof(int)); /* inizializzazione valida a run-time */
```

[1] Come new, creatori e distruttori tipici dei linguaggi ad oggetti.

```
  int *q, *r = &z;                  /* due puntatori, uno vale l'.ind. di z */

  *q = y;          /* errore: il puntatore non è stato assegnato */
  *r = y;          /* corretto: r è l'indirizzo di z, quindi assegno z */
  if (x) return p; /* corretto: la memoria allocata rimane disponibile */
  else return &z;  /* errore: all'esterno di f non posso usare z */
}
```

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.

```
funzione()
{
  auto int x=10; /* inizializzata */
  auto int y;
  /* non inizializzata: il valore
   * che assume è del tutto casuale
   */;

  /* di fatto il programma poteva
   * essere scritto in modo
   * equivalente
     int x, y;
     x=10;
   * auto può generalmente essere
   * omesso
   * si noti 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, 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 può essere chiamata solo in questo file */
{
  int j; /* allocata sullo stack */
  j = -i; /* variabili locali, "i" è l'argomento della funzione  */
  return j;
}

int count(void) /* count è 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 dimostra una serie di dichiarazioni di variabili statiche/dinamiche, locali/globali. L'esempio è abbondantemente commentato.

Appartengono quindi alla classe `static` questa classe 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 cnt = 0;
  fprintf(stderr, "** debug: cnt=%d v=%d\n",
    cnt++, v);
}
```

10.7 Variabili extern

Vengono poste automaticamente nella classe di memorizzazione `extern` classe 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.

```
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.

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

```
int a=1;
char dollaro = '\$';
long y= 10000 * 20000;

long w = y * 2;
  /* solo variabili locali */
```

Vettori e strutture possono essere inizializzati solo se globali.

```
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. 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

Le righe che iniziano con il carattere # sono comandi dati al preprocessore

La direttiva #define viene usata per definire delle *macro* che possono avere eventualmente dei parametri

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-sostituirè 'testo-da-sostituirè va dallo spazio dopo nome fino alla fine della linea; può continuare su linee successive se l'ultimo carattere della linea è \ (fa ignorare il carattere di a capo al precompilatore)

Per esempio

```
#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", 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

```
#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:

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 <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 nella directory corrente.

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, ad esempio:

In generale è buona regola non mettere negli header-file il codice delle funzioni, ma solo la loro definizione.

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
```

Tutti i caratteri compresi tra `#if` e `#endif` vengono inclusi 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" è già stata definita
         */
```

```
#endif

#ifndef macro
        /*
         * questo codice viene considerato
         * solo se "macro" non è 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 (in fase di debugging per esempio):

```
#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 binari, al contrario dei file di testo, non sono leggibili correttamente con i classici editor di testo come emacs e vi, oppure con comandi come cat, more, less, ecc. Questo perchè i file binari non contengono solo i caratteri ASCII stampabili, come caratteri alfanumerici e segni di interpunzione, ma possono contenere *tutti* i caratteri ASCII.

E' 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 binario 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 bytes 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 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 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)

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.

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 necessati permessi.

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.

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` indirizzo di memoria in cui scrivere (leggere)
- `size` dimensione del dato singolo

- `nelem` numero di dati da trattare
- `f` è il puntatore a `FILE` che identifica il file in cui scrivere (leggere)

Il programma seguente effettua la copia un file binario di interi.

```
/*
 * copia un file binario di interi
 *
 * i file binari non sono leggibili correttamente
 * con editor classici e comandi come cat, more ...
 */

#include <stdio.h>

int main(int argc, char ** argv)
{
  int buf[1024];
  int n;
  FILE *fin, *fout;

  if (!(fin = fopen(argv[1], "r"))) {
    perror(argv[1]);
    exit(1);
  }
  if (!(fout = fopen(argv[2], "w"))) {
    perror(argv[2]);
    exit(2);
  }
  do {
    n = fread(buf, sizeof(int), 1024, fin);
    fwrite(buf, sizeof(int), n, fout);
  } while (n);

  fclose(fin);
  fclose(fout);
  return 0;
}
```

Il programma dichiara 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` con le due linee di codice

```
  if (!(fin = fopen(argv[1], "r")))
  if (!(fout = fopen(argv[2], "w")))
```

che aprono rispettivamente un file in lettura e uno in scrittura. I nomi dei file vengono otenuti dalla linea di comando, e sono rispettivamente il secondo e il terzo parametro[1]. Le istruzioni

[1] Il primo parametro della linea di comando è sempre il nome del programma lanciato.

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` legge un numero di elementi diverso da 0.

Con l'istruzione

```
n = fread(buf, sizeof(int), 1024, fin);
```

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(buf, sizeof(int), n, fout);
```

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`. E' 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 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 sulla funzione `fgets`[2] 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, ...);
```

e il suo funzionamento è esattamente identico a quello della `scanf`[3], 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 in Figura 12.1 mostra l'utilizzo della coppia di funzioni `fgets` e `sscanf`:

[2] 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.

[3] Attenzione alla "s" in più!

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

#define MAX_STUD  (5)

struct studente {
  char nome[80];
  int voto;
};

struct studente stud[MAX_STUD];

int main()
{
  char s[100];
  int cont = 0;
  int err = 0;
  int somma = 0;

  while ((fgets(s, sizeof(s), stdin)) && (cont < MAX_STUD)) {
    sscanf(s, "%80s %d", stud[cont].nome, &stud[cont].voto);
    if (stud[cont].voto >= 18 && stud[cont].voto <= 30) {
      somma += stud[cont].voto;
      cont++;
    } else err++;
  }

  if (cont > 0)
    somma /= cont;

  printf("studenti %d, media %d errori %d\n", cont, somma, err);

  return 0;
}
```

Figura 12.1: Il programma `voti.c`.

Il programma legge delle stringhe di testo con `fgets` dalla console (il 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`, 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` (End Of File), che da tastiera equivale ad immettere il carattere Ctrl-D[4]. Nel ciclo 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 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 di 5 elementi.

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

#define N (5)

int main()
{
  char s[100];
  float m[N][4];
  int i;

  i = 0;
  while (fgets(s, sizeof(s), stdin)) {
    sscanf(s, "%f %f %f %f", &m[i][0], &m[i][1], &m[i][2], &m[i][3]);
    printf("%f %f %f %f\n", m[i][0], m[i][1], m[i][2], m[i][3]);
    i++;
  }

  return 0;
}
```

Da notare che il programma non effettua controlli sul fatto che vengano lette più righe di quelle memorizzabili nella matrice.

12.8 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. Le funzioni `fscanf` e `fprintf` hanno il medesimo

[4]Tenere premuti contemporaneamente i tasti Control e D.

funzionamento: la differenza consiste nel fatto che è possibile specificare su quale file leggere e scrivere (anche i file standard `stdin` e `stdout`.

Le due funzioni sono infatti 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 (e 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.

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

#define N (5)

int main()
{
  float m[N][4];
  int i;

  i = 0;
  while (!feof(stdin)) {
    fscanf(stdin, "%f %f %f %f",
      &m[i][0], &m[i][1], &m[i][2], &m[i][3]);
    fprintf(stdout, "%f %f %f %f\n",
      2 * m[i][0], 2 * m[i][1], 2 * m[i][2], 2 * m[i][3]);
    i++;
  }

  return 0;
}
```

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.

12.9 Esempio di lettura di matrici con `fgets` e `sscanf`

I programmi illustrati nelle sezioni 12.7 e 12.8 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 colonna. 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.

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

#define MAX_N (10)

int main()
{
  char s[100], *s1;
  float m[MAX_N][MAX_N];
  int i, j;

  i = 0;
  while (fgets(s, sizeof(s), stdin)) {
    j = 0;
    if ((s1 = strtok(s, " ")) != NULL) {
      sscanf(s1, "%f", &m[i][j]);
      printf("%f ", m[i][j]);
      j++;
    } else {
      printf("errore nel parsing dei valori\n");
      return 1;
    }
    while ((s1 = strtok(NULL, " ")) != NULL) {
      sscanf(s1, "%f", &m[i][j]);
      printf("%f ", m[i][j]);
      j++;
    }
    printf("\n");
    i++;
  }

  return 0;
}
```

Il programma legge le stringhe dal file di ingresso con l'istruzione `fgets`, corrispondenti ad una riga della matrice. Per isolare gli elementi di ciascuna colonna, viene usata la funzione di libreria `strtok`. 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, riceve in ingresso la stringa da decomporre in token (`s` nell'esempio), e il separatore dei token (uno spazio, nell'esempio). Ogni successiva chiamata a `strtok` deve passare `NULL` come stringa, poichè questa è stata specificata nella prima chiamata. Ecco perchè nella chiamata ad `strtok` presente nel ciclo `while` 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 all'avanzare del numero di riga e colonna.

12.10 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.

E' 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).

E' 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 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 genrico.

12.10.1 Il programma `count.c`

Il programma `count.c` conta il numero di caratteri che vengono letti dallo standard input.

```
#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

```
  while (fgets(s, sizeof(s), stdin))
```

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`. 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, poichè 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 l'output viene scritto sul file `output.dat`.

12.10.2 Un altro esempio di redirezione da linea di comando

Supponendo che il programma di Figura 12.1 generato di chiami `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`.

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:

```
$ 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.

12.11 `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

```
#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.

```
$ ./tscanf
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:

```
$ ./tscanf
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:

```
$ ./tscanf
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

```
#include <stdio.h>

int main()
{
```

```
  char s[100];
  int n1, n2, n3, n4, n5, n6;

  fgets(s, sizeof(s), stdin);
  sscanf(s, "%d %d %d", &n1, &n2, &n3);
  printf("valori inseriti: %d %d %d\n", n1, n2, n3);

  fgets(s, sizeof(s), stdin);
  sscanf(s, "%d %d %d", &n4, &n5, &n6);
  printf("valori inseriti: %d %d %d\n", n4, n5, n6);

  return 0;
}
```

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:

```
$ ./tfgets
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:

```
$ ./tfgets
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 bubber 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

```
{
  char buff[BUFSIZ], st[BUFSIZ];

  if ((fgets(buff, sizeof(buff), fin) == NULL) ||
      sscanf (buff, "%s", st) != 1)
  {
    /* ... */
  }
}
```

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

Utilizzo di 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.

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:

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

```
int main() {

  printf("%s: %s %d\n", __FILE__, __FUNCTION__, __LINE__);
  func1();

  printf("%s: %s %d\n", __FILE__, __FUNCTION__, __LINE__);
  func2();

  return 0;
}
```

Il file `func1.c` è il seguente:

```
#include <stdio.h>

int func1() {
  printf("%s: %s %d\n", __FILE__, __FUNCTION__, __LINE__);

  return 0;
}
```

Il file `func2.c` è il seguente:

```
#include <stdio.h>

int func2() {
  printf("%s: %s %d\n", __FILE__, __FUNCTION__, __LINE__);

  return 0;
}
```

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;

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 (non contengono la funzione `main`!), da cui la segnalazione di errore.

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.

13.1 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.

La condizione che permette a `make` di decidere se un file deve essere ri-generato o meno si basa sul tempo di creazione e modifica dei file, che viene aggiornato automaticamente dal sistema operativo in seguito ad un accesso in scrittura dei file.

`make` rigenera il file target solo se il tempo di modifica di uno o più 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.

13.2 Esempio di dipendenze

Si supponga per esempio che il programma eseguibile

```
prog
```

sia realizzato come in Sezione 13, ovvero sia composto dai tre file `prog.c`, `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`.

Ciascun file oggetto dipende a sua volta dal rispettivo file sorgente.

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` e `prog`. Quindi un nuovo building ri-genererà soltanto `func2.o` e `prog`, mentre `func1.o` può essere recuperato dalla precedente compilazione.

13.3 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.

E' 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
  gcc -o prog prog.c func1.o func2.o

func1.o: func1.c
  gcc -c -o func1.o func1.c

func2.o: func2.c
  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`. 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[1]. 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

[1] Esistono anche le cosiddette *librerie dinamiche*, ma la loro trattazione va oltre gli scopi di queste dispense.

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 in 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[2].

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 prevere 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 del tipo -lNOME di linking, 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
```

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.

E' anche possibile verificare il contenuto di una libreria, sempre utilizzando il comando ar. Digitando per esempio il comando:

[2]Talvolta la libreria matematica, come altre, vengono linkate per default dal linker.

```
# ar t /usr/lib/libm.a
```

si ottiene il seguente output (una parte):

```
...
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.

E' 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
```

E' 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. E' 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.h`

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.

La funzione `printf` permette di eseguire un output di dati di vari tipi, formattato secondo specifiche definibili in una opportuna stringa di formato.

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 14.3.4).

14.3.1 Ricerca e ordinamento

La libreria standard implementa due funzioni molto utili per la manipolazione di insiemi omogenei di dati: la funzione `bsearch` implementa l’algoritmo di ricerca binaria in modo generalizzato, ovvero applicabile a qualsiasi insieme di dati. Lo stesso vale per la funzione `qsort`, la quale implementa in maniera generica l’algoritmo di ordinamento QuickSort.

Le funzioni sono dichiarate come segue:

- `void *bsearch ( void *key, void *base, size_t nmemb, size_t size, compar_fn_t compar);`
  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;

- `void qsort (void *base, size_t nmemb, size_t size, compar_fn_t compar);`
  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;

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. Per questo motivo in ciascuna funzione deve essere 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.

Allo stesso modo, potendo lavorare su insiemi arbitrari di dati, si deve specificare quanti sono tali elementi.

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 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, poichè il confronto dipende dal significato dei bit che compongono gli elementi. Per esempio, il confronto tra due interi è banale e viene fatto dal compilatore, così come quello tra numeri a virgola mobile. 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).

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.

14.3.2 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;
- `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.

14.3.3 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.

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 di 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.

14.3.4 Generazione di numeri casuali

Tra le funzioni, ci sono quelle per la generazione di numero casuali (pseudo-casuali, in realtà) 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[3].

La funzione `rand()` ritorna un numero pseudo-casuale compreso nell'intervallo [0, RAND_MAX]. E' semplice ottenere un numero pseudo-casuale in un qualsiasi intervallo [a, b] utilizzando opportunamente l'operatore modulo (vedi Sezione 6.19), come nell'esempio seguente:

```
valore = a + rand() % (b - a + 1);
```

Il generatore di numero 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 in Figura 14.1 illustra il funzionamento del generatore di numeri pseudo-casuali per simulare il lancio di due dadi.

Il programma legge da linea di comando il seme e il numero di lanci da effettuare, rispettivamente in questo ordine. Se i parametri non vengono forniti, sono usati i valori di default.

[3] 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.

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

int main(int argc, char **argv)
{
  int a, b, seme = 0, n = 1, i;

  switch (argc) {
    case 3 : sscanf(argv[2], "%d", &n);
    case 2 : {
      sscanf(argv[1], "%d", &seme);
      break;
    }
    case 1 : break;
    default : return;
  }

  srand(seme);

  for (i = 0; i < n; i++) {
    a = 1 + rand() % 6;
    b = 1 + rand() % 6;
    printf("%d %d\n", a, b);
  }

  return 0;
}
```

Figura 14.1: Simulazione del lancio di due dadi.

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.

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 prefernza stilistica non è molto importante, quanto invece lo è la coerenza in tutto il codice sorgente.

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 8 spazi[1].

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 definite creatori e distruttori per gli 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.

E' 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 lasciate decidere al programma principale come gestire l'errore.

Commentare bene il codice: in linea di massima, evitare costrutti particolarmente *furbi*, che possano 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 i termini di licenza nel file sorgente. 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.

[1] Attenzione alla configurazione predefinita dell' editor che potrebbe essere scorretta.

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.

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 sono alla base di molte tecniche per la gestione delle informazioni saranno presentate in questo capitolo.

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. Una *funzione ricorsiva* è la funzione che richiama sè stessa.

La ricorsione è una tecnica molto utile poichè 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.)

Per esempio il seguente programma 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.

```
/*
 * fibonacci.c
 * Calcolo dell'i-esimo numero di Fibonacci
 * Il valore di i viene letto da linea di comando
 */
#include <stdio.h>
#include <stdlib.h>

long fibonacci(long n)
{
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

int main(int argc, char **argv)
{
  long n;

  if (argc != 2) return 1;

  if (n = atoi(argv[1]))
    printf("fibonacci(%d) = %d\n", n, fibonacci(n));

  return 0;
}
```

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.

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.

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. E' 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 in un sistema operativo
- la pila, o stack

  - serve a memorizzare i dati locali e gli indirizzi di ritorno delle subroutine
  - serve a risolvere equazioni scritte in forma polacca inversa

- la doppia coda

  - ...

- l'array, nella forma di vettore, si usano 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 impiegata 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 che 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.

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 tue tipi. Le *operazioni globali* riguardano tutti gli elementi della lista, mentre le *operazioni locali* riguardano i singoli elementi della lista.

Esempi di operazioni globali sono:

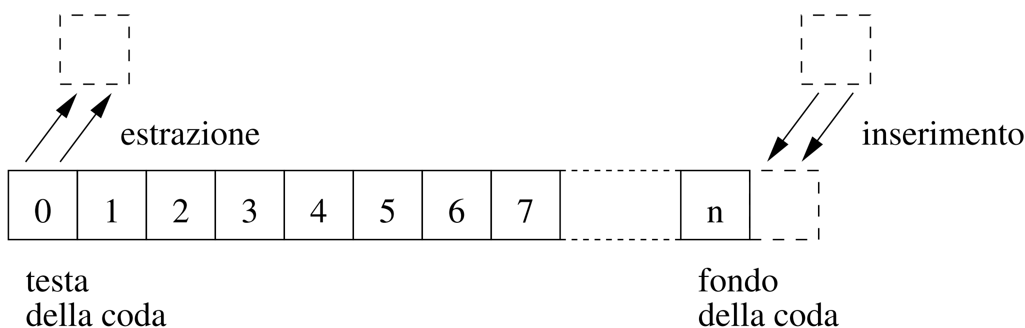

Figura 17.1: Le operazioni che si possono attuare su una coda.

- 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.

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.

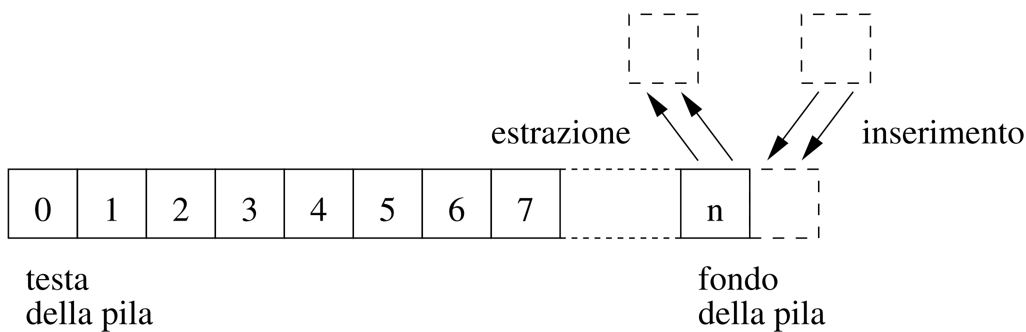

Figura 17.2: Push e pop: le principali operazioni che si compiono su uno stack.

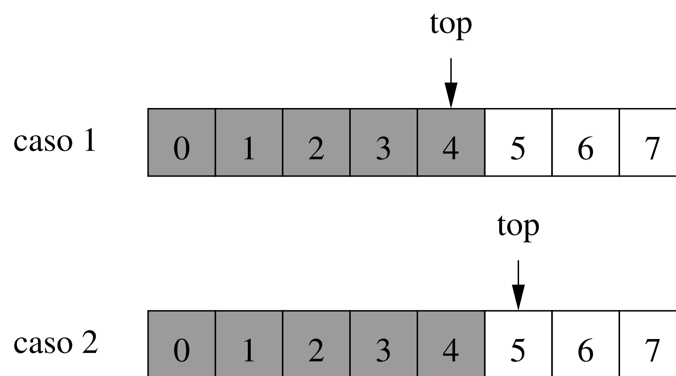

Figura 17.3: Due differenti implementazioni di uno stack.

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 risolvere equazioni 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.

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 (vedi Figura 17.4).

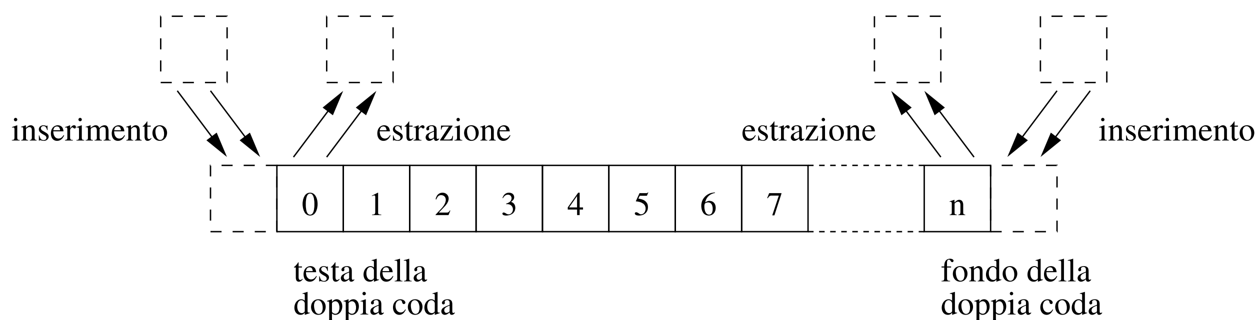

Figura 17.4: Le operazioni che si possono attuare su una doppia coda.

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.

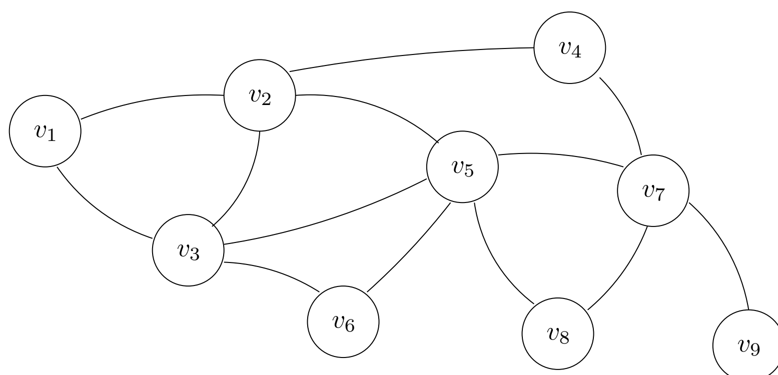

Figura 17.5: Esempio di grafo connesso composto da 9 nodi.

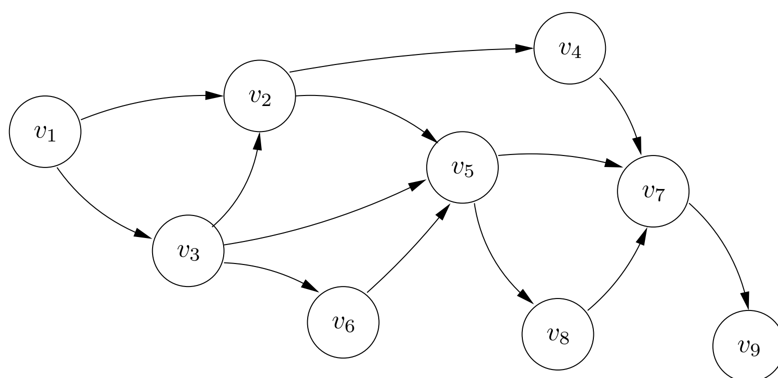

Figura 17.6: Esempio di grafo diretto composto da 9 nodi.

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 se' stesso

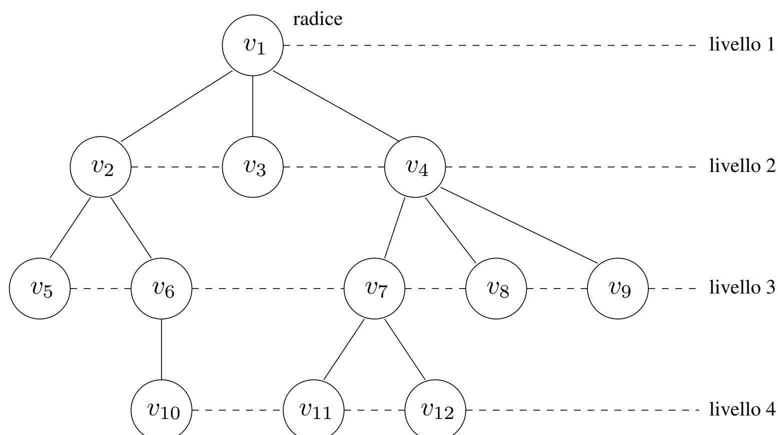

Figura 17.7: Esempio di albero.

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 poichè 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.

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 efficienti 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:

$$v_1 \quad [[v_2 \quad ((v_5) \quad (v_6 \quad (v_{10})))] \quad [v_3] \quad [v_4 \quad (v_7 \quad ((v_{11}) \quad (v_{12})) \quad (v_8) \quad (v_9))]]$$

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:

$$v_1 \quad v_2 \quad v_5 \quad v_6 \quad v_{10} \quad v_3 \quad v_4 \quad v_7 \quad v_{11} \quad v_{12} \quad v_8 \quad v_9$$

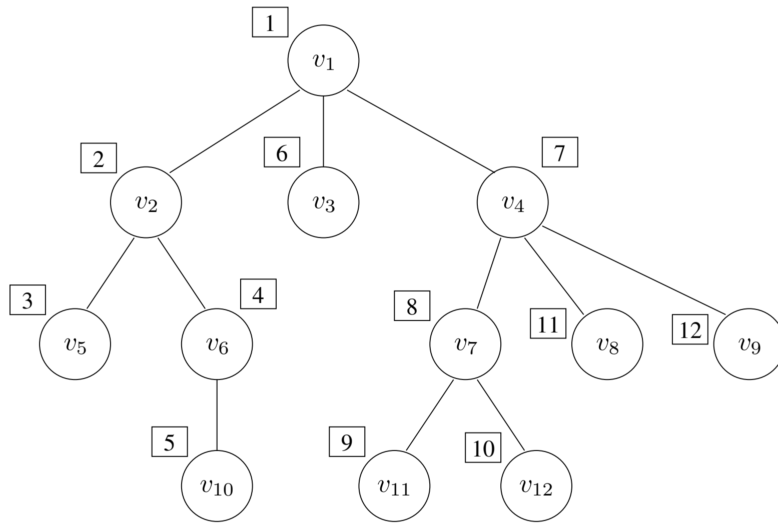

Figura 17.8: Visita anticipata dell'albero di Figura 17.7.

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:

$$[[((v_5) \quad (v_6 \quad (v_{10}))) \quad v_2] \quad [v_3] \quad [(((v_{11}) \quad (v_{12})) \quad v_7) \quad (v_8) \quad (v_9)] \quad v_4] \quad v_1$$

Anche in questo caso, le parentesi sono utilizzate per evidenziare i sottoalberi. Senza l'indicazione dei sottoalberi la scrittura diviene:

$$v_5 \quad v_6 \quad v_{10} \quad v_2 \quad v_3 \quad v_{11} \quad v_{12} \quad v_7 \quad v_8 \quad v_9 \quad v_4 \quad v_1$$

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

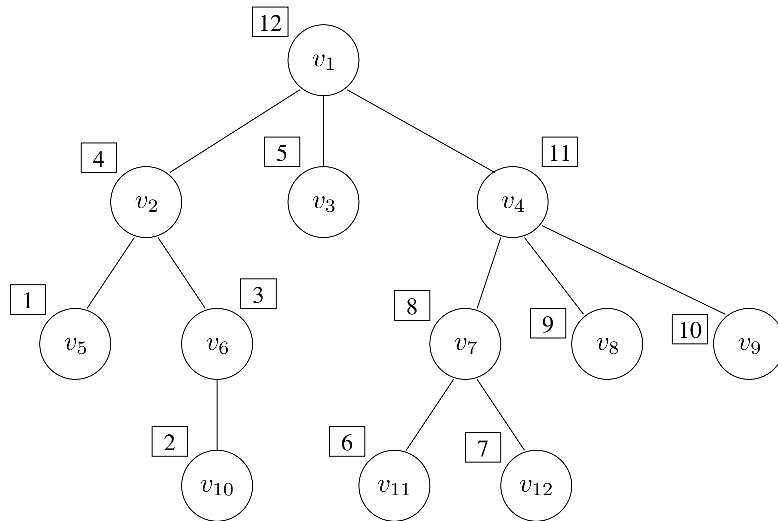

Figura 17.9: Visita differita dell'albero di Figura 17.7.

- 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.

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 sequente:

$$v_7 \quad v_4 \quad v_8 \quad v_{11} \quad v_2 \quad v_6 \quad v_5 \quad v_3 \quad v_9 \quad v_{12} \quad v_6 \quad v_{10}$$

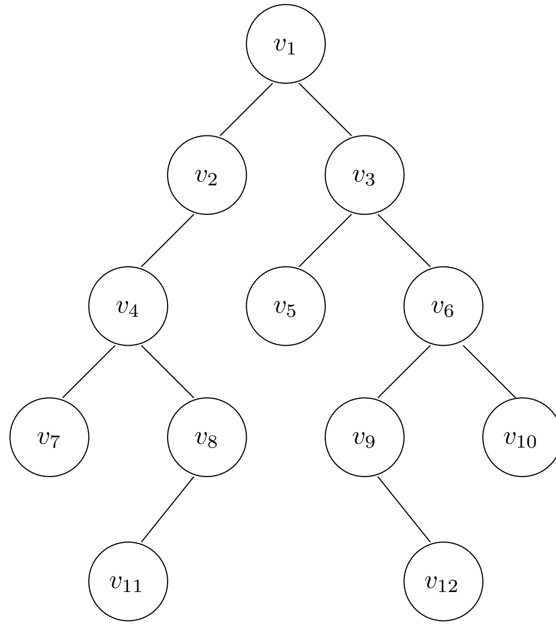

Figura 17.10: Esempio di albero binario.

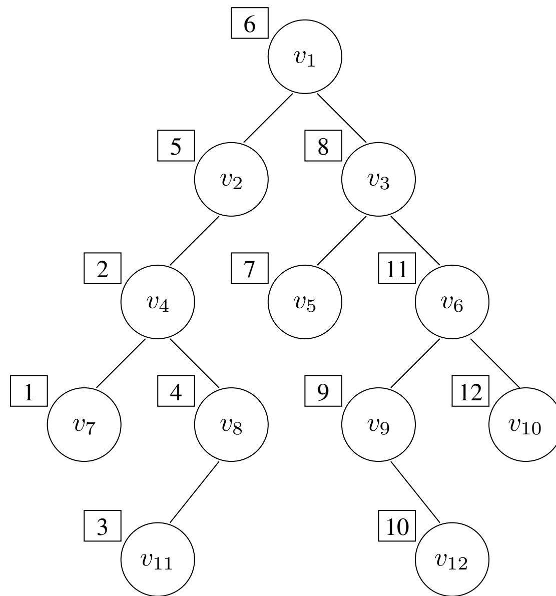

Figura 17.11: Visita in ordine simmetrico dell'albero binario di Figura 17.10.

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 sulla 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

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 adiacenti in memoria disposti ad indirizzi crescenti; ciascun elemento può richiedere una o più celle;
- tipicamente tutti gli elementi hanno la medesima lunghezza, cioè contengono lo stesso numero di celle

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 richieste da ciascun elemento, ovvero la dimensione dell'elemento

Di conseguenza l'indirizzo del primo elemento della struttura sequenziale è:

$$addr(x_0) = addr_b$$

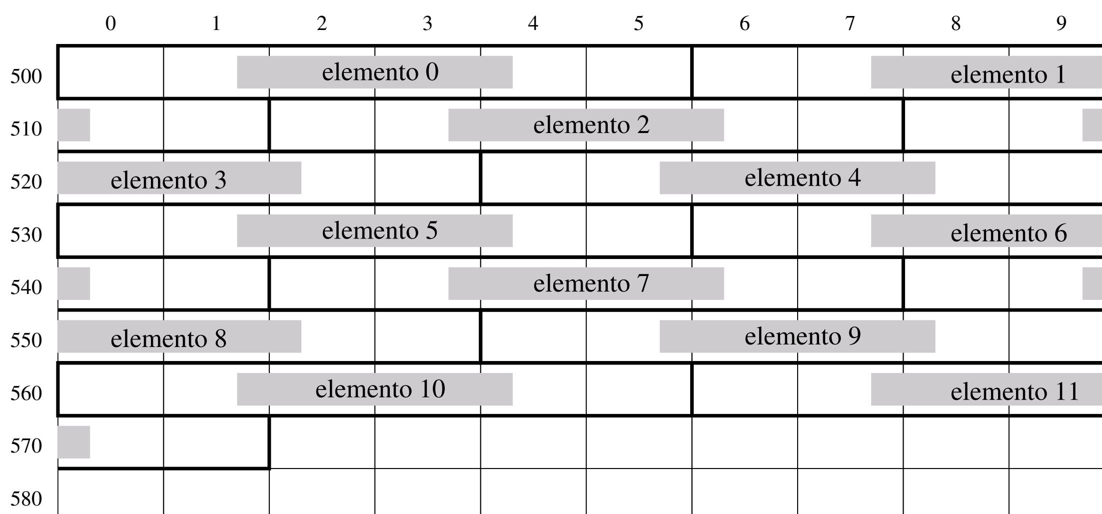

Figura 18.1: Esempio di memorizzazione di una struttura sequenziale.

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 di 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. 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 abbiamo 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)

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

	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.

sulla lunghezza media dei blocchi: quanto più la media si avvicinerà alla lunghezza massima, tanto più sarà conveniente l'uso della prima tecnica.

E' possibile valutare la convenienza nell'uso 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

$$mem_1 = dm$$

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 la cancellazione di tutti gli elementi che lo devono seguire e la loro scrittura in una posizione più avanti
- analogo discorso vale per l'eliminazione, se non si vogliono lasciare celle inutilizzate

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.

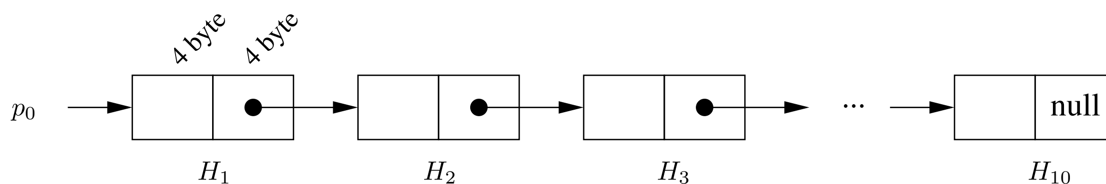

Figura 18.3: Esempio di lista.

	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.

18.2 Esempio: la coda circolare

18.3 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, purchè 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 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

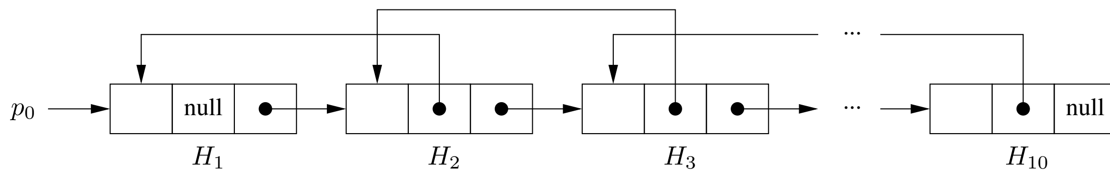

Figura 18.5: Esempio di lista bidirezionale.

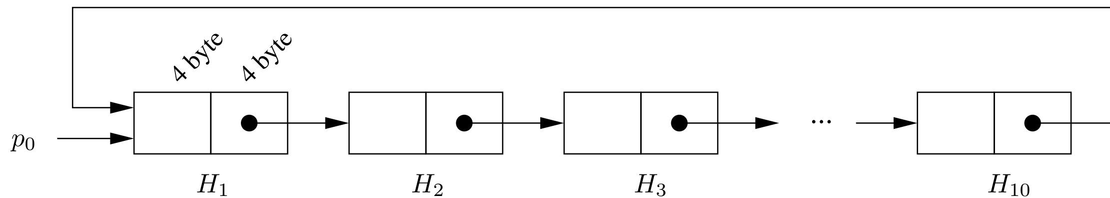

Figura 18.6: Esempio di lista circolare.

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.

E' 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.

18.3.1 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

$$ptr(H_{i-1}) \leftarrow addr(H_i)$$

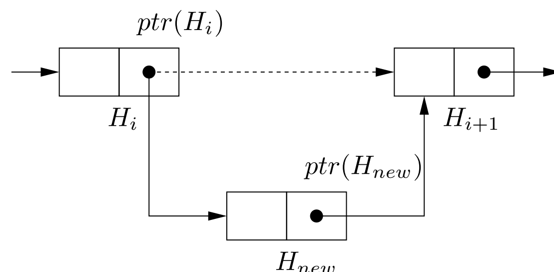

Figura 18.7: Inserimento dell'elemento H_{new} in una lista.

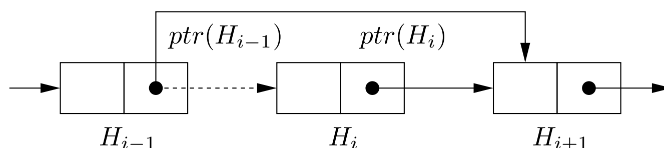

Figura 18.8: Eliminazione dell'elemento H_i dalla lista.

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 ne' raggiungibile.

E' 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.

18.4 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.

18.5 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.

18.6 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

18.7 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 (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)

18.8 Ricerca sequenziale

Gli elementi nella tabella sono allocati senza seguire alcuna regola di ordinamento.

L'operazione di ricerca si effettua scandendo gli elementi della tabella fino al reperimento della chiave desiderata.

è una tecnica scarsamente efficiente in cui il tempo medio di ricerca è pari a N/2, se N è la lunghezza della tavola.

Si può migliorare il tempo di accesso concentrando gli elementi in cui si prevede un accesso frequente nelle prime posizioni.

Una tavola su cui si opera una ricerca sequenziale può essere memorizzata come struttura sequenziale o a catena.

18.9 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.

Il tempo massimo di ricerca è $log_2 N$.

E' una tecnica di ricerca veloce, ma è difficile inserire nuovi elementi (riordino delle chiavi).

Una tavola su cui si opera una ricerca binaria deve essere memorizzata come struttura sequenziale con elementi di lunghezza costante, dato che l'accesso deve avvenire tramite calcolo dell'indirizzo.

18.10 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 difficile prevedere l'inserimento di nuovi elementi (le relative chiavi devono ancora permettere l'identificazione di indirizzi diversi).

Alta velocità di accesso, ma scarsa applicabilità a causa delle ipotesi restrittive.

18.11 Accesso calcolato (hashing)

E' 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'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: problema della collisione di elementi (le chiavi sono dette sinonimi).

Nasce il problema della scelta delle posizioni nella struttura sequenziale corrispondenti agli elementi in collisione.

La scelta della funzione di accesso ed il criterio di gestione delle collisioni determinano la bontà dell'implementazione della tavola.

I sinonimi possono essere memorizzati in catene interne o esterne alla tavola.

18.12 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.

18.13 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.

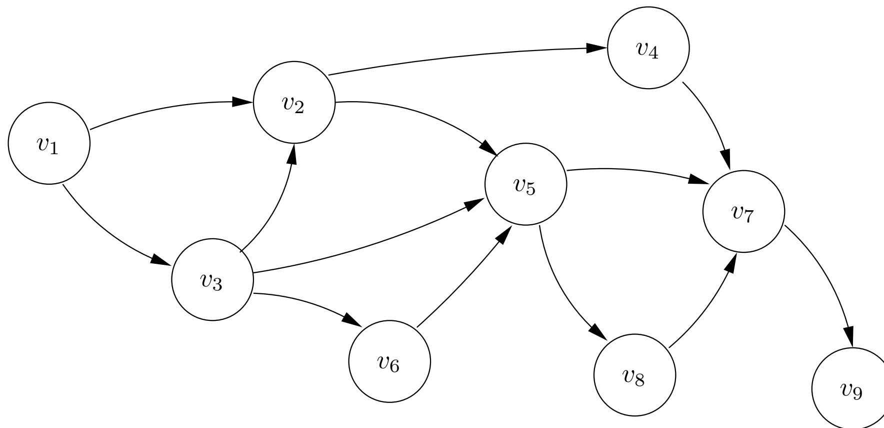

Figura 18.9: Grafo di esempio.

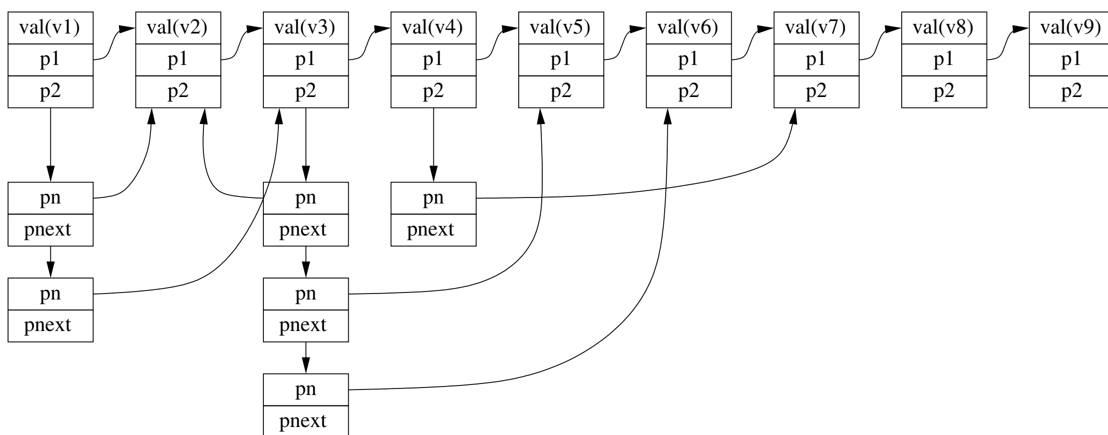

Figura 18.10: Esempio di memorizzazione del grafo di Figura 18.9 (sono visualizzati soltanto una parte dei collegamenti).

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 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.

19.1.1 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 finchè 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.

```
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 controlla la fine del ciclo. Invece di effettuare il test sul numero di elementi esaminati, si controlla l'uguaglianza col valore sentinella. 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 $n/2$.

19.1.2 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 di elementi.

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, se no 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[1].

```
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'è */
```

[1] La funzione `bsearch1` è così denominata, invece che nel modo più naturale di `bsearch`, poichè nella libreria standard `stdlib` è presente una versione più generica dell'algoritmo di ricerca.

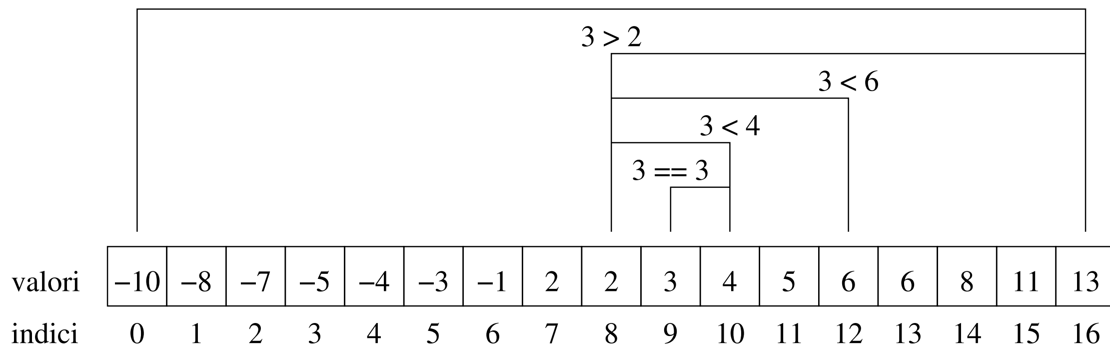

Figura 19.1: Esempio di ricerca binaria.

```
  } 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 */
  }
}
```

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 $\log(n) + 1$ 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 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.

19.2.1 Bubblesort

Uno dei più semplici esempi di algoritmo di ordinamento è il seguente algoritmo, il *bubble sort*[2]:

```
void bubblesort(int l[], int n)
{
  int i, j;

  for (i = 0; i < n; i++) {
    for (j = i+1; j < n; j++) {
  if (l[i] > l[j]) {
  swap(&l[i], &l[j]);
      }
    }
  }
}
```

Vengono in questo caso scanditi tutti gli elementi del vettore e vengono confrontati con *tutti i seguenti*, e in caso il test dica che l'ordinamento non è quello desiderato, i valori vengono scambiati. Così facendo, i valori vengono man mano spostati rispettando l'ordinamento reciproco.

[2]Vedi file di esempio `bubblesort.c`.

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 $Ds = ab'\check{}\,a'b$, $Dx = cb'\check{}\,c'b$ e $Dy = ac'\check{}\,a'c$. 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: poiché 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 è %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.

Appendice A

Tabella degli operatori

Tabella A.1: Tutti gli operatori del C.

Symbol	Name or Meaning	Associativity
	Highest Precedence	
( )	Function call	⟹
[ ]	Array element	
.	Structure or union member	
− >	Pointer to structure member	
!	Logical NOT	⟸
∼	One's complement	
-	Unary minus	
++	Increment	
−	Decrement	
&	Address	
*	Indirection	
(type)	Type cast [for example, (float) i]	
sizeof	Size in bytes	
*	Multiply	⟹
/	Divide	
%	Remainder	
+	Add	⟹
-	Subtract	
<<	Left shift	⟹
>>	Right shift	
<	Less than	⟹
<=	Less than or equal to	
>	Greater than	
>=	Greater than or equal to	
==	Equal	⟹
!=	Not equal	
&	Bitwise AND	⟹
^	Bitwise exclusive OR	⟹
|	Bitwise OR	⟹
&&	Logical AND	⟹
||	Logical OR	⟹
? :	Conditional	⟸
=	Assignment	⟸
*=, /=, %=, +=, -=, <<=, >>=, &=, ^ =, | =	Compound assignment	
,	Comma	⟹
	Lowest Precedence	

Appendice B

Il compilatore gcc

Esistono vari compilatori C disponibili sia sia open che closed source, sia gratuiti che a pagamento. Inoltre sono disponibili per pressochè 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.

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
```

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

Bibliografia

[Amb08] Steve Ambler, *Some useful linux commands*, http://www.er.uqam.ca/nobel/r10735/unixcomm.html (ultimo accesso: 23 luglio 2008).

[pro08] The GNU project, *Bash reference manual*, http://www.gnu.org/software/bash/manual/bashref.html (ultimo accesso: 13 ottobre 2008).

[Rub07a] Alessandro Rubini, *Dispense di C online*, http://gnudd.com/srt-2007/A-C-X.html (2007).

[Rub07b] ———, *Dispense di C online*, http://gnudd.com/srt-2007/A-C-X-more.html (2007).

Indice analitico

albero, 153
albero binario, 155
albero libero, 153
albero ordinato, 153
algoritmo, 19
ar, 132, 133
archi, 151
array, 76
auto, 100

Big-endian, 61
break, 44, 46
bubble sort, 172
bug, 22

cammino, 152
cammino semplice, 152
casting, 57, 81
cat, 119, 121
catena, 162
cd, 15
chiave, 151
ciclo, 152
circuito, 152
classe di memorizzazione, 97
coda, 149
compilatore, 27
contatore, 41
continue, 46
cos, 132
count.c, 119

debugging, 22
dichiarazione, 29, 71
dipendenza, 128
directory, 8
do-while, 39
doppia coda, 150

EOF, 112, 116
esecuzione, 22
eseguibile, 25
exp, 132
extern, 102

fclose, 110, 112, 114
feof, 117
fflush, 110, 112
fgets, 110, 114, 116–120, 122–125
FIFO, 149
FILE, 110–113
file oggetto, 27
file oggetto, 27
foglia, 153
fopen, 110, 111, 113, 120
for, 40
fprintf, 110, 116, 119
fputs, 110, 119
fread, 110, 112, 114
fscanf, 110, 116, 117, 119, 122, 124, 125
funzione, 90
funzione ricorsiva, 144
fwrite, 110, 112

gets, 114
goto, 47
grado, 154
grafo, 151

header, 27

identificatore, 32
if, 42
if-else, 42
indentazione, 31, 142
indici, 151
int, 72
interprete, 25

lati, 151
libc, 131
libc.a, 132

libreria, 90
lifetime, 97
linker, 27
lista, 162
lista bidirezionale, 163
lista circolare, 163
lista lineare, 148
lista multipla, 164
Little-endian, 61
loader, 27
loop, 38
ls, 10
lvalue, 68

macro, 105
main, 94
make, 128
makefile, 129
man, 15
mascheratura, 64, 65
matrice, 151

nodi, 151
nodi adiacenti, 152
NULL, 111, 119

operatori, 48

parola chiave, 32
parole riservate, 32
passaggio per riferimento, 92
passaggio per valore, 91
percorso, 152
pila, 149
pop, 149
printf, 116, 117, 119, 132
processore, 19
programma, 19, 21
programmazione, 19
prototyping, 90
puntatore, 76
push, 149
pwd, 15

radice, 153
register, 100
ricerca sequenziale, 169
ricorsione, 144
rm, 13

scanf, 114, 116, 117, 119, 122, 124
scope, 96
semantica, 22
sh, 15
shell, 6
sin, 132
sintassi, 22
sizeof, 114, 116, 125
sorgente, 25
sottoalbero, 154
sqrt, 132
sscanf, 114, 116, 117, 122
stack, 149
static, 101
stderr, 11, 110
stdin, 110, 114, 116, 117, 119, 120
stdio.h, 110–112
stdout, 110, 117, 119
storage class, 97
stringa, 148
strtok, 118, 119
struttura sequenziale, 159
strutture dati astratte, 146
strutture dati concrete, 146
switch, 44

tabella, 151
tavola, 151, 165
testing, 23
tipizzazione, 29, 71
typedef, 81

union, 80

variabile, 33
variabili globali, 96
variabili locali, 96
vertici, 151
vettore, 76, 151
visibilità, 96
visita, 154
visita in ordine anticipato, 154
visita in ordine differito, 155
visita in ordine simmetrico, 156

warnings, 30
while, 38, 119