Conoscere ed usare

Linguaggio ANSI C

Guida alla programmazione dei sistemi embedded

Antonio Di Stefano

INWARE EDIZIONI

Antonio Di Stefano

Conoscere ed usare

Linguaggio ANSI C

Guida alla programmazione dei sistemi embedded

INWARE EDIZIONI

Titolo:

CONOSCERE ED USARE
Linguaggio ANSI C

Prima Edizione - Maggio 2006
ISBN 88-901665-2-5

Autore:

Antonio Di Stefano

Copyright:

© 2005 – INWARE Edizioni S.r.l.
Via Cadorna 27/31
20032 Cormano (MI)

Tel: 02-66504755 Fax: 02-66508225
info@inwaredizioni.it www.inwaredizioni.it

Stampa:

Graficonsult - MI

Indice

Prefazione

L'ANSI C è oggi il linguaggio in assoluto più utilizzato per la programmazione dei sistemi a microprocessore. Questo primato deriva in primo luogo dalle sue caratteristiche di efficienza e compattezza del codice generato, ma anche dalla sua grande versatilità: utilizzando il C è facile adattare il proprio codice alle caratteristiche della macchina che dovrà eseguirlo, così come utilizzare un livello di astrazione molto più spinto quando richiesto.
Queste caratteristiche hanno fatto sì che il linguaggio C venisse sempre più spesso utilizzato per la programmazione di sistemi a microcontrollore e sistemi embedded[1] in genere, rimpiazzando quasi completamente l'uso dell'assembler.
Questo libro si colloca proprio in questo contesto: esso vuole fornire un'introduzione sia al linguaggio ANSI C, sia, soprattutto, al suo utilizzo per la programmazione dei sistemi embedded. Se infatti è abbastanza comune trovare degli ottimi manuali che descrivono il linguaggio C in se, sono pochissimi i testi che descrivono le tecniche e gli accorgimenti che occorre adottare quando il linguaggio viene utilizzato per programmare macchine dotate di risorse molto limitate (in termini di memoria, di velocità di esecuzione e spesso anche di energia disponibile per l'esecuzione dei compiti).
In particolare il libro si rivolge sia a chi ha già una certa esperienza con i sistemi a microprocessore o microcontrollore e con l'uso dell'assembler e vorrebbe iniziare ad utilizzare il linguaggio C per la loro programmazione, sia a chi conosce già il linguaggio C e vorrebbe avvicinarsi alla programmazione dei sistemi embedded, pur non avendo nessun esperienza diretta in questo campo. Alla prima categoria di lettori il libro mostrerà, oltre alla sintassi di base del linguaggio, una serie di tecniche utilizzate ad alto livello per la gestione e l'elaborazione dei dati, tecniche di debug ed ottimizzazione e molti accorgimenti utili per la scrittura del codice. Ai secondi verrà mostrata, più che la sintassi del linguaggio, la prospettiva corretta e le tecniche più appropriate da adottare quando si programmano sistemi dedicati, che per loro natura richiedono un controllo ed un'interazione molto stretta tra programma e macchina.
Nella scelta degli argomenti trattati si è preferito fornire una visione sufficientemente completa sui diversi aspetti coinvolti nella programmazione dei sistemi embedded, piuttosto che descriverne solo alcuni in maniera esaustiva. Anche la descrizione del linguaggio C, sebbene sufficientemente completa, è limitata agli elementi fondamentali: da questo punto di vista il testo non sostituisce un manuale di riferimento sul C.
La prima parte del libro descrive il linguaggio C nei suoi aspetti fondamentali (sintassi, istruzioni, costrutti, etc.), la seconda parte affronta invece argomenti più specialistici: verranno spiegati i metodi più corretti per scrivere ed organizzare programmi e progetti complessi, le strutture dinamiche e la loro implementazione, alcuni algoritmi comuni, l'aritmetica fixed point e la sua applicazione all'elaborazione dei segnali, l'uso delle interruzioni, le tecniche di debug ed ottimizzazione del codice e l'impiego di sistemi operativi e microkernel. Nonostante i diversi argomenti siano presentati in modo relativamente semplice ed intuitivo, è utile per il lettore possedere le nozioni basilari sul funzionamento dei sistemi a microprocessore e sulla scrittura di un programma.

Antonio Di Stefano

(1) Piccoli sistemi a microcontrollore o microprocessore dotati di tutte le risorse hardware e software necessarie per svolgere una o più funzioni specifiche in maniera autonoma.

1. Il linguaggio C

BREVE STORIA DEL LINGUAGGIO C

Il linguaggio C nasce nel 1972 nei laboratori BELL di proprietà AT&T per mano di Dennis Ritchie, come linguaggio ad alto livello per la scrittura di sistemi operativi. Il suo primo uso estensivo, infatti, coincide con la riscrittura dall'assembler del sistema operativo UNIX. Nel 1978 Brian Kernighan e Dennis Ritchie pubblicarono un testo diventato ormai una leggenda: "The C Programming Language" (figura 1.1 e 1.2) nel quale veniva fornita la prima definizione ufficiale del linguaggio C. Alla versione K&R del linguaggio C (dal nome dei rispettivi autori) si susseguirono una nutrita serie di varianti e di dialetti per poter programmare in C o pseudo C qualsiasi sistema a microprocessore.
Gli stessi linguaggi Java, C++, C#, JavaScript e PHP oggi diffusissimi per la programmazione su PC possono essere considerati delle varianti estese del linguaggio C originale. All'epoca della comparsa del C non esistevano sistemi operativi o architetture hardware standard a diffusione planetaria come accade oggi (il primo PC apparve nel 1981). La possibilità di far girare un programma su sistemi diversi era affidata soltanto alla cosiddetta “portabilità del codice” ovvero la possibilità di ricompilare un codice sorgente con piccole modifiche sui diversi sistemi. Nel 1983 l’American National Standard Institute (ANSI), rico-

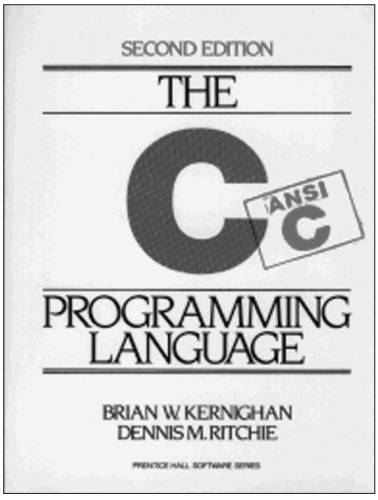


Figura 1.1
L’ edizione originale del libro “The C Programming Language”

noscendo il notevole interesse suscitato dal linguaggio C e la necessità di una maggiore uniformità tra le diverse versioni, avviò il processo di standardizzazione del linguaggio. Il risultato fu la definizione dello standard ANSI C, che è attualmente considerata la versione "ufficiale" del linguaggio. La nascita dello standard rafforzò l'interesse di molti programmatori e produttori, che iniziarono a creare compilatori per diverse macchine, in grado di riconoscere ed interpretare sorgenti scritti in ANSI C, facilitando così ulteriormente la portabilità del codice. Altri produttori si specializzarono nello sviluppo dei cosiddetti *cross compiler* ovvero di compilatori in grado di generare da un sorgente C un codice assembler da far girare su una macchina diversa rispetto a quella usata per effettuare la compilazione. Nell'ultimo decennio infine, grazie alla diffusione dei microcontrollori ed al perfezionamento della tecnologia dei compilatori, il linguaggio C si è imposto anche come il linguaggio ad alto livello in assoluto più utilizzato per la programmazione di questi dispositivi ed in genere di piccoli sistemi a microprocessore, grazie alla sua capacità di ottenere un codice macchina molto compatto ed efficiente.

LA PORTABILITÀ DEL CODICE NELLA REALTÀ

Il fatto che il linguaggio C sia disponibile per qualsiasi macchina, non significa che, scritto un programma per un determinato sistema operativo o un determinato microprocessore, sia sufficiente una ricompliazione per eseguire lo stesso programma su qualsiasi sistema. Il C, infatti, garantisce una portabilità dei soli costrutti del linguaggio e di un insieme minimo di funzioni di libreria denominate "funzioni di libreria standard". Si immagini ad esempio un codice che ha come unica funzione quella di visualizzare la scritta "Ciao!" sul video. Probabilmente, una volta compilato con i vari tool disponibili, funzionerà sia su Windows che su Linux, come su Mac o Palm OS, ma probabilmente non funzionerà affatto su un microcontrollore PIC, per il semplice motivo che il PIC non ha un video! È possibile però collegare al PIC un display LCD ed intervenire sulla routine utilizzata dal programma per visua-

Figura 1.2
Alcune delle moltissime edizioni del libro di K&R

lizzare il testo in modo da ottenere lo stesso risultato su LCD.
Questo banale esempio mostra che uno stesso codice è tanto più facilmente portabile quanto più esso è indipendente dall'hardware su cui verrà eseguito.
D'altro canto è anche chiaro che non è conveniente programmare piccoli sistemi, dotati di risorse molto limitate, nello stesso modo in cui vengono programmati sistemi più grandi e complessi (quali sono ad esempio i PC). Per ottenere buone prestazioni non è possibile infatti prescindere dalle caratteristiche dell'hardware che eseguirà il codice.
La portabilità e l'ottimizzazione del codice sono pertanto due esigenze abbastanza contrapposte, che sono però entrambe ben gestibili dal linguaggio C. Esso infatti permette di scrivere indifferentemente codice a più alto livello (più indipendente dalla macchina e quindi più portabile) o codice ad un livello più basso, che utilizza delle istruzioni e degli operatori adatti alla manipolazione dei dati e della memoria ad un livello molto vicino all'assembler (quindi più adatto a sfruttare meglio le caratteristiche dell'hardware). Quest'ultima caratteristica è quella che ha decretato il grande successo del C nel campo della programmazione dei sistemi embedded ed è la stessa che rende il linguaggio C molto interessante dal punto di vista dalla presenta trattazione. Nei successivi capitoli infatti verrà data particolare enfasi a questo aspetto.

IL PRIMO PROGRAMMA

Per iniziare l'analisi del linguaggio, si consideri il seguente programma:

```
#include <stdio.h>

main() {
  printf("Hello world!\n");
}
```

Questo piccolo programma, tratto dal già citato libro di Kernighan & Ritchie, ha il solo scopo di visualizzare sullo schermo il testo "Hello world!".
Le parti che lo costituiscono sono abbastanza tipiche ed è possibile ritrovarle anche in programmi più complessi. All'inizio di ogni programma in genere sono presenti delle direttive (`#include` in questo caso) per indicare al compilatore quali librerie utilizzare durante la compilazione. In questo caso viene richiamata soltanto una libreria standard (*stdio.h*) che contiene le più importanti funzioni per l'input e l'output di dati (il cosiddetto "Standard Input/Output"). Come già accennato compilando il programma su un PC lo standard output è costituito dallo schermo, mentre lo standard input è la tastiera (su sistemi privi di questi dispositivi entrambi potrebbero essere rappresentati ad esempio da una porta di comunicazione seriale). Dalla seconda riga inizia la funzione "`main`". In un programma C il codice è normalmente raggruppato in una o più "funzioni", che possono essere richiamate con un meccanismo simile a quello delle *subroutine* dell'assembler (o di altri linguaggi). La funzione "`main`" ha un ruolo particolare perché è la funzione richiamata automaticamente all'avvio del programma, quindi contiene il codice che deve essere eseguito inizialmente. Le eventuali altre funzioni sono richiamate dal codice che si trova all'interno della funzione `main`. In questo caso nella `main` è presente una sola istruzione ("`printf`"), che è in realtà una funzione della libreria *stdio.h*, utilizzata per stampare stringhe, caratteri o numeri sullo standard output. I dati che verranno stampati sono

racchiusi tra virgolette. Il carattere "`\n`" alla fine della stringa serve per specificare un "a capo" dopo la stringa stessa.
Maggiori dettagli su tutti questi aspetti verranno forniti nei prossimi capitoli.
Si consideri adesso un esempio leggermente più complesso:

```
#include <stdio.h>

main() {
  int a, b, som, prod;

  a = 6;
  b = 5;
  som = a + b;
  prod = a * b;

  printf("Somma = %d \n", som);
  printf("Prodotto = %d \n", prod);
}
```

Compilando ed eseguendo il codice, il risultato sullo schermo sarà:

```
Somma = 11
Prodotto = 30
```

La struttura del codice è simile a quella dell'esempio precedente, le differenze risiedono nelle istruzioni contenute nella funzione `main`. In questo caso inizialmente vengono dichiarate quattro variabili intere (`a`, `b`, `som` e `prod`), successivamente viene assegnato un valore a due di queste, viene calcolato il valore della loro somma e del loro prodotto ed il risultato stampato a schermo.
Al contrario di altri linguaggi a più alto livello (ad esempio il Basic), in C è necessario dichiarate esplicitamente tutte le variabili utilizzate ed il loro tipo (in caso contrario si ottiene un errore!). Per fare questo, all'inizio della funzione viene indicato il tipo voluto, seguito dall'elenco delle variabili. In questo caso sono state impiegate variabili intere, quindi è stato utilizzato il tipo `int`.
Per visualizzare il valore dei risultati viene utilizzata ancora la funzione `printf`. Questa versatile funzione, permette di inserire nel testo da visualizzare anche il valore di alcune variabili, semplicemente inserendo nella stringa il simbolo "%d" (nel caso di numeri interi) nel punto in cui si desidera visualizzare il valore ed elencare di seguito, in ordine di apparizione, le variabili da stampare.
Considerando i due piccoli esempi riportati si possono osservare alcuni particolari che hanno validità generale nel linguaggio C. Il C è un linguaggio case sensitive, cioè fa differenza tra lettere maiuscole e lettere minuscole. Ad esempio per il compilatore `Printf` (con la P maiuscola) è una funzione diversa da `printf` (con la p minuscola), che tra l'altro non esiste a meno di non definirla appositamente. Analogamente scrivere `MAIN` non è equivalente a scrivere `main`. Lo stesso vale anche per le variabili e qualsiasi altro elemento sintattico. È possibile combinare arbitrariamente lettere maiuscole e minuscole nei nomi dati a variabili e funzioni; avendo cura di utilizzare gli stessi identici nomi in tutto il codice. Altro elemento sintattico importante è il punto e virgola (;) che deve essere posto alla fine di ogni riga di codice e serve per separare in modo uni-

voco le diverse istruzioni. È molto facile dimenticare un punto e virgola scrivendo molte righe. Questo errore normalmente è segnalato dal compilatore, che tuttavia a volte può fraintendere completamente il senso del codice e dare una serie di errori incomprensibili senza apparenti spiegazioni. In questi casi l'origine degli errori è quasi sempre un punto e virgola dimenticato…
La sintassi delle funzioni (come ad esempio la `main`) consiste in un identificativo (cioè il loro nome) seguito dagli eventuali parametri, che sono indicati tra parentesi tonde. Il codice della funzione è invece delimitato da parentesi graffe. L'uso e le caratteristiche delle funzioni verranno trattate più dettagliatamente in uno dei prossimi capitoli.

Il compilatore Dev-C++

Il metodo migliore per imparare un linguaggio di programmazione è quello di provare in prima persona a scrivere del codice, scontrarsi con gli errori riportati dal compilatore, constatare come le modifiche apportate influenzano i risultati. Per familiarizzare con il linguaggio C è conveniente iniziare ad utilizzare un compilatore su un comune PC. Dal momento che il codice compilato funzionerà sempre su PC, è possibile verificare facilmente il comportamento dei programmi scritti ed eventualmente usare degli strumenti che facilitano l'individuazione di eventuali errori (*debugger*).
Una volta acquisita familiarità con la sintassi del linguaggio, scrivere un programma per un qualsiasi altro sistema risulterà estremamente semplice ed in molti casi si potranno perfino riutilizzare le routine già scritte. Esistono molti compilatori per PC, con caratteristiche diverse per tipo d'interfaccia, modalità di utilizzo, tipo di codice oggetto che riescono a produrre e prezzo. Molti compilatori ad esempio, anche di ottimo livello (come lo storico GCC), funzionano a "linea di comando", cioè richiedono che l'utente richiami i vari elementi che li costituiscono utilizzando il prompt dei comandi. Altri offrono invece un ambiente di sviluppo integrato (IDE), cioè un'unica interfaccia grafica da cui è possibile eseguire tutte le operazioni in maniera molto intuitiva: scrittura del codice, compilazione, linking, esecuzione e debug. Questi ultimi risultano quindi più intuitivi da usare, soprattutto per chi è alle prime armi. Un compilatore dotato di questa caratteristica, è il Bloodshed Dev-C++, che ha tra i vari aspetti positivi anche quello di essere completamente gratuito (freeware). È possibile trovare il programma su Internet, scaricandolo all'indirizzo www.bloodshed.net.
Una volta installato il software è possibile provare i programmi riportati in questo capitolo ed improvvisarne di nuovi. L'ambiente grafico è suddiviso in diverse zone (figura 1.3), in cui è possibile scrivere il codice, visualizzare la struttura del progetto (cioè i file utilizzati), lancia-

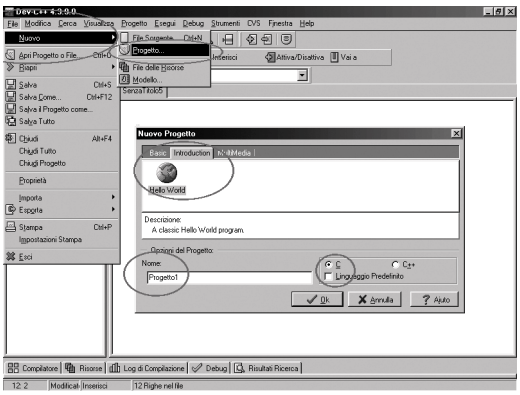


Figura 1.3
Creazione di un nuovo progetto

re i comandi, visualizzare i messaggi del compilatore ed eseguire il debug. Per provare a compilare ed eseguire i programmi presentati occorre invocare dal menu File\Nuovo la voce Progetto (seguire la figura 1.3).
A questo punto si aprirà una finestra da cui è possibile selezionare diversi modelli di progetti predefiniti. Se non diversamente specificato, nella presente trattazione verrà usato "Hello Wolrd", che si trova nel tab Introduction. Selezionato questo, inserire il nome da dare al nuovo progetto e selezionare a destra "C" come linguaggio.
Per "progetto" si intende un file (utilizzato dell'IDE) che raccoglie tutte le impostazioni ed i file utilizzati nella stesura di un programma. Chiaramente i file di programma veri e propri saranno altri (nel caso specifico ad esempio "Hello.c", visibile a sinistra sotto il folder col nome del progetto, in figura 1.3). Dopo avere dato l'OK ed avere indicato una directory in cui memorizzare i file, verrà visualizzato un piccolo programma di esempio che sarà sostituito dal nuovo codice.
Non è strettamente necessario definire un nuovo progetto per potere provare un programma, per semplicità è anche possibile utilizzare un singolo file contente il codice C (creato selezionando la voce "Nuovo File sorgente" dal menu File\Nuovo).

La compilazione e l'esecuzione del programma avviene utilizzando i due appositi pulsanti (fig. 1.4) o le corrispondenti voci del menu Esegui.
All'avvio dell'esecuzione viene aperta automaticamente una finestra in cui viene eseguito il programma, che si chiude al termine dell'esecuzione. Per rendere visibili i risultati è consigliato l'uso della funzione `getchar`, contenuta nella libreria *stdio.h*, che blocca l'esecuzione del programma in attesa della pressione di un tasto. È sufficiente quindi inserire prima della chiusura del `main` la chiamata alla funzione:

```
…
   getchar();
}
```

Per visualizzare l'output del programma senza bisogno di aggiungere del codice, basta eseguire il programma ottenuto (che si chiamerà *NomeProgetto.exe*) dal prompt del DOS. Normalmente anche i programmi scritti per sistemi embedded non terminano, ma sono scritti in modo da ripetere le operazioni all'infinito.

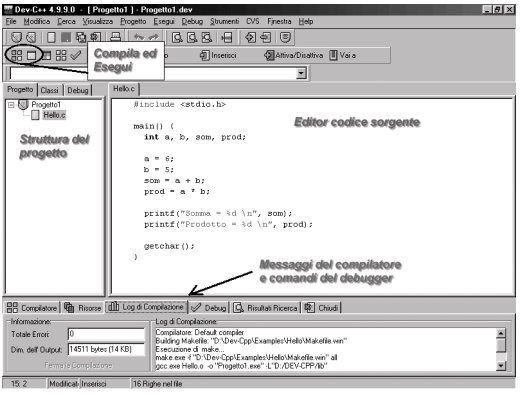


Figura 1.4
Disposizione degli strumenti nell'ambiente DEV-C++

2. Variabili e tipi di dati

USO DELLE VARIABILI

Normalmente in un programma è necessario manipolare e memorizzare dei dati, che possono provenire dall'esterno (da un interfaccia utente, da un file, da un dispositivo hardware), possono essere i risultati intermedi delle elaborazioni eseguite o possono essere utilizzati come supporto per le elaborazioni stesse. Per gestire questi dati nel linguaggio C, così come negli altri linguaggi ad alto livello (Basic, Pascal…), vengono utilizzate le variabili. In C una variabile è rappresentata, come accennato nel capitolo precedente, da un identificativo scelto dal programmatore, opportunamente dichiarato all'interno del codice. A differenza dei linguaggi a più alto livello, il C adotta internamente un meccanismo simile a quello che si utilizza in assembler per gestire le variabili, al punto che è perfino possibile fare in modo che una variabile sia memorizzata in un registro del processore piuttosto che nella memoria! Anche per questo motivo è necessario dichiarare esplicitamente nel programma tutte le variabili che si intendono utilizzare e definirne il tipo. I diversi tipi di dati sono caratterizzati da una diversa occupazione di memoria e sono gestiti in maniera differente dal compilatore (ed anche dall'hardware in alcuni casi).
È possibile suddividere i tipi di dati predefiniti gestiti dal linguaggio C in due categorie: tipi interi e tipi in virgola mobile. I primi sono utilizzati in genere per esprimere numeri interi positivi e negativi, i secondi per gestire numeri reali (o non interi in generale).
Ad esempio possono essere rappresentati con tipi interi numeri quali 3, 1768, -245673987; mentre richiedono l'utilizzo di tipi in virgola mobile numeri come 0.54337, -230.356, 87.1e+6.
Queste due classi di variabili sono gestite dal C con tecniche completamente diverse e per questo motivo non è sempre possibile applicare agli uni tutti gli operatori che si possono applicare agli altri.

TIPI INTERI

L'elenco dei tipi interi predefiniti è riportato nella tabella 2.1, in cui è indicato anche il numero di bit richiesti da ciascun tipo e l'intervallo numerico rappresentabile con ciascuno di essi.
Si noti che tutti i tipi sono disponibili in due versioni: una "normale" (*signed*), che permette di rappresentare sia numeri positivi che negativi ed una chiamata *unsigned*, con cui possono essere espressi soltanto numeri positivi, però in un intervallo doppio rispetto alla versione con

segno. Il tipo intero più piccolo è il `char` e una variabile di questo tipo corrisponde ad un'occupazione di memoria di un byte.

TIPO	NUMERO DI BIT	INTERVALLO RAPPRESENTABILE
char	8	da -128 a 127
unsigned char	8	da 0 a 255
short int	16	da –32768 a 32767
unsigned short int	16	da 0 a 65535
int	16 (32)	da –32768 a 32767 (vedi *long*)
unsigned int	16 (32)	da 0 a 65535 (vedi *unsigned long*)
long (int)	32	da –2147483648 a 2147483647
unsigned long (int)	32	da 0 a 4294967295

Tabella 2.1 *Tipi interi predefiniti*

Il nome di questo tipo è dovuto al fatto che esso permette di esprimere un numero compreso tra 0 e 127 (o 255), quindi può essere utilizzato per rappresentare dei caratteri alfanumerici, indicandoli attraverso il loro codice ASCII. Oltre che per memorizzare piccoli numeri e caratteri, i `char` possono essere convenientemente utilizzati per gestire generiche grandezze ad 8 bit, quali piccoli contatori e dati relativi a periferiche hardware (valore di una porta di I/O, un campione sonoro, il valore di un canale di colore RGB, etc.).
Ecco un esempio sull'uso del tipo `char`:

```
#include <stdio.h>

main() {
  char a;

  a = 68;

  printf("a = %d \n", a);
  printf("a = %c \n", a);

  a = a + 1;

  printf("a = %d \n", a);
  printf("a = %c \n", a);

  getchar();
}
```

Nel programma è dichiarata una variabile di tipo `char` a cui è stato assegnato un valore numerico (68), successivamente questo valore è visualizzato sullo schermo utilizzando la funzione `printf`.
Nel primo caso (usando il simbolo "%d") viene visualizzato il valore numerico intero associato alla variabile, nel secondo (usando "%c") viene visualizzato il carattere ASCII corri-

spondente (“D” in questo caso). I caratteri %d e %c utilizzati nella funzione `printf` svolgono quindi la funzione di convertire il valore numerico in un output testuale, interpretando il dato secondo un certo codice. Il valore numerico della variabile ovviamente è sempre 68 in entrambi i casi. Per dimostrare questo, alla riga successiva la variabile è incrementata di uno ed il contenuto di nuovo stampato.
I tipi `int` sono sicuramente i più utilizzati, in quanto permettono di rappresentare un intervallo numerico abbastanza esteso, con un’occupazione di memoria ragionevole e comunque variabile a seconda delle esigenze.
Ad esempio il tipo `short int` richiede soltanto due byte (16 bit) ed ha il pregio di essere facilmente gestito dalla maggior parte di architetture hardware (anche ad 8 bit). Il tipo `int` (senza altra specificazione) ha una lunghezza non ben definita, che può essere di 16 bit o 32 bit, a seconda del compilatore utilizzato e soprattutto della macchina per cui viene compilato il codice. Per evitare sorprese è quindi opportuno consultare il manuale del proprio compilatore per accertarsi di questo dato.
Si può notare comunque che in un caso l’`int` diventa in pratica un duplicato dello `short int`, mentre nell’altro del `long int`. Per facilitare la portabilità del codice è consigliato l’uso di questi ultimi. Il tipo long int (abbreviato di solito in `long`) utilizza 32 bit, quindi permette di rappresentare numeri interi abbastanza grandi (dell’ordine dei miliardi) e può essere comodo per non incorrere in overflow durante calcoli con numeri interi o per memorizzare dei dati allineati nativamente a 32 bit (ad esempio indirizzi di memoria o dati su un bus).
È molto importante avere un’idea precisa sulla grandezza dei dati gestiti, prima di iniziare a scrivere il codice. Il C lascia infatti al programmatore assoluta libertà nella gestione dei tipi interi (come già si è detto essi possono rappresentare anche grandezze non numeriche) e quindi non viene generato alcun errore se in un calcolo si eccede la capacità di rappresentazione del tipo.
Questo aspetto è mostrato nell’esempio seguente, in cui il risultato di un’operazione supera la capacità di rappresentazione della variabile utilizzata:

```
#include <stdio.h>

main() {
   short int a=1453, b=1097, rsi;
   long lc=a, ld=b, rl;

   rsi = a*b;
   rl = lc*ld;

   printf("%d x %d = \n", a, b);
   printf("%d (short int)\n", rsi);
   printf("%d (long)", rl);

   getchar();
}
```

Nell’esempio sono state dichiarate due variabili `short int` e due `long`, a cui è stato assegnato lo stesso valore (inizializzato proprio in corrispondenza della loro dichiarazione). Il valore del loro prodotto non è rappresentabile con uno `short int` (infatti è maggiore di 32767), quindi il valore ottenuto è errato, pur non avendo nessuna segnalazione di errore. Usando il `long` invece si ottiene il risultato corretto.

TIPI IN VIRGOLA MOBILE

Con il termine "virgola mobile" (*floating point* in inglese) viene fatto riferimento ad una particolare rappresentazione di un numero tramite una mantissa (una base eventualmente decimale) ed un esponente. Si tratta in pratica di un metodo simile alla più comune notazione scientifica. Ad esempio il numero intero 1234 può essere espresso in virgola mobile come 1.234e3, come 0.076 può essere espresso come 7.6e-2 (dove "e" sta per "x10 elevato a"). In entrambi i casi si è spostata la virgola (da qui il nome) e modificato l'esponente. Dal momento che è possibile gestire separatamente la base e l'esponente (sia in valore che in segno), è possibile rappresentare un intervallo di valori molto più grande di quello rappresentabile con i tipi interi, che comprende anche i numeri non interi ed è anche possibile mantenere la stessa precisione sia che si operi su numeri molto piccoli che molto grandi.
I tipi in virgola mobile previsti dall'ANSI C sono quelli mostrati in tabella 2.2.

TIPO	NUMERO DI BIT	INTERVALLO RAPPRESENTABILE
float	32	da 3.4e-38 a 3.4e38
double	64	da 1.7e-308 a 1.7e308
long double	80	da 3.4e-4932 a 1.1e4932

***Tabella 2.2** Tipi in virgola mobile gestiti dal linguaggio C*

Rispetto alle variabili intere, la maggiore differenza tra un tipo ed un altro risiede in questo caso non tanto nell'intervallo di valori rappresentabili, ma nella precisione con cui si riesce a rappresentare un numero, in termini di cifre significative dopo la virgola. In particolare il tipo `float` consente una precisione di circa 7 cifre, il tipo `double` circa 15 e il tipo `long double` ben 18.
I numeri in virgola mobile per le loro caratteristiche sono particolarmente indicati per applicazioni scientifiche o per l'esecuzione di algoritmi che non tollerano bene la perdita di precisione che si verifica utilizzando i numeri interi. Nella pratica questo tipo di rappresentazione è usata spesso per comodità, ogni volta che occorre utilizzare numeri non interi, pur non essendo necessaria una grande precisione. È bene tenere presente però che l'esecuzione di calcoli in virgola mobile è estremamente più lenta rispetto a quella basata su numeri interi (soprattutto su macchine non dotate di coprocessore matematico), consuma una maggiore quantità di memoria e per questo non è idonea per l'impiego su piccoli microprocessori e microcontrollori.
L'uso dei numeri float nel codice è simile a quello dei numeri interi:

```
#include <stdio.h>

main() {
   float r=5.74, pi=3.14159, c;

   c = 2*pi*r;

   printf("Circonferenza = %f \n", c);

   getchar();
}
```

Il programma mostra un esempio di calcolo del valore di una circonferenza dato il raggio. Da notare l'uso del carattere "%f" nella `printf`, necessario per convertire correttamente i numeri in virgola mobile in testo.

ESEMPIO

Il programma seguente esegue la conversione di una temperatura da gradi Fahrenheit a gradi Celsius, richiedendo all'utente il dato da convertire:

```
#include <stdio.h>

main() {
   float far, cel;

   /* immissione dati */
   printf("Temperatura in gradi F = ");
   scanf("%f", &far);

   /* conversione */
   cel = 5*(far-32)/9;

   /* stampa */
   printf("Temperatura Celsius = %f\n", cel);
}
```

Sono state utilizzate due variabili di tipo `float` per memorizzare la temperatura in gradi Fahrenheit e Celsius. Le linee di testo che si trovano racchiuse tra i simboli /* e */ nel programma sono dei commenti e sono utilizzati, come in altri linguaggi, per facilitare la lettura e la comprensione del codice. Nei commenti è possibile scrivere del testo qualsiasi, anche su più righe. Per i commenti su una sola linea è possibile usare anche la doppia barra "//" (questa sintassi non è prevista dallo standard ANSI, ma è supportata dalla maggior parte dei compilatori C\C++).
Per richiedere il dato all'utente è stata utilizzata la funzione `scanf`. Questa funzione è la complementare di `printf` ed ha lo scopo di leggere un dato dallo "Standard Input", cioè dalla tastiera (nel caso del PC). La funzione attende che l'utente immetta del testo e prema il tasto Enter; finché non succede questo l'esecuzione del programma rimane ferma. A questo punto il dato testuale viene convertito nel formato specificato (in questo caso `float`, indicato da "%f") e viene caricato nella variabile indicata come secondo argomento della funzione. Notare il carattere "&" che precede il nome della variabile. La sua funzione verrà spiegata in seguito, per ora basti sapere che si deve sempre aggiungere alle variabili all'interno della funzione `scanf`.
Ottenuto il dato, viene eseguita la conversione utilizzando la nota formula, che implica delle comuni operazioni aritmetiche. Va precisato a questo punto che gli operatori aritmetici di base messi a disposizione dal C sono: + (somma), - (differenza), * (prodotto), / (quoziente) e % (modulo, cioè resto della divisione).
Questi operatori funzionano indifferentemente con numeri interi o float, tranne l'ultimo, che può essere utilizzato solo con variabili intere.

RAPPRESENTAZIONE INTERNA

Come vengono rappresentati in memoria i vari tipi di dati utilizzati? Sebbene non sia indispensabile conoscere questo particolare per potere programmare in C, può essere utile avere un'idea un po' più precisa, sia per capire se si sta eseguendo una operazione potenzialmente errata, sia per avere un maggiore controllo sui dati memorizzati in memoria, sui registri o sui file.
Si considerino i numeri interi. È stato visto che ciascun tipo intero utilizza una certa quantità di bit per rappresentare un numero. Un particolare importante è che la rappresentazione che viene utilizzata è il binario naturale per gli interi *unsigned* ed il binario naturale in complemento a 2 per i tipi *signed*.
È possibile verificare questo con il seguente programma:

```
#include <stdio.h>

main() {
  unsigned char b;

  b = -1;

  printf("Valore = %d", b);
}
```

Il valore visualizzato è 255! Questo perché –1 in complemento a 2 viene espresso con un numero binario che ha tutti i bit impostati ad 1, che nel caso di un byte interpretato come unisgned significa proprio 255. Nello stesso modo, se `b` vale 255, sommando 1, si ottiene 0. Il linguaggio C non considera errori questo tipo di operazioni, quindi non segnala un overflow, lasciando al programmatore assoluta libertà nella manipolazione del dati.
Assegnando una variabile di tipo `short int` (16 bit) ad una di tipo `char` (8 bit), nel char verranno copiati solo gli 8 bit meno significativi. Il compilatore segnala in questi casi che la conversione porta alla perdita dei bit più significativi, tuttavia sapendo (in base all'algoritmo implementato) che lo `short int` non assume valori maggiori di 255, si ha la garanzia che la conversione non causerà degli errori. Lo stesso meccanismo vale per la conversione dai `long` agli altri tipi più piccoli. Ovviamente la conversione nella direzione opposta non crea problemi, il compilatore copia i bit meno significativi ed aggiunge in testa degli zeri (o 1, nel caso di numeri negativi).
Per i tipi floating point il meccanismo è più complesso; in questo caso normalmente viene utilizzata la rappresentazione standard IEEE-754, che nel caso del tipo `float` (32 bit) prevede di utilizzare 1 bit per il segno della mantissa, 23 bit per la mantissa ed 8 per l'esponente (codificato con scostamento 128, cioè 0 significa –128, 129 significa 1 e così via). Proprio per questa maggiore complessità e per il fatto che in molti casi questi dati devono essere gestiti da hardware apposito (il coprocessore matematico), l'uso dei numeri a virgola mobile è gestito con maggiore severità dal compilatore che, oltre a segnalare un maggior numero di errori, impedisce di manipolare manualmente i valori.
Quando non è disponibile un coprocessore matematico il compilatore aggiunge al programma le proprie routine per "emularlo", cioè consentono lo svolgimento dei calcoli via software con un notevole incremento dei tempi di esecuzione.

IL CASTING

Il termine “casting” indica la conversione di un numero da un tipo ad un altro. Questo si può verificare nel caso di semplici assegnazioni, ma anche durante la valutazione del valore di espressioni aritmetiche composte da variabili di tipi diversi. Il casting è gestito automaticamente dal compilatore e normalmente è trasparente per l’utente, esistono però delle circostanze in cui non tenere conto delle conversioni da effettuare può portare a risultati errati.
Si consideri ad esempio la riga di codice che converte la temperatura da gradi Fahrenheit a Celsius nel programma di esempio visto in precedenza:

```
cel = 5*(far-32)/9;
```

Se fosse stata utilizzata un’espressione del tipo:

```
cel = 5/9*(far-32);
```

il risultato sarebbe stato errato, pur essendo le due espressioni matematicamente identiche. Sia la variabile `cel` che `far` sono di tipo `float`, mentre gli altri numeri riportati nella formula sono invece degli interi (il compilatore li interpreta come `float` solo se hanno un punto decimale: ad esempio 5.0 e 9.0). Le operazioni vengono eseguite con le consuete precedenze tra gli operatori e procedendo da sinistra a destra. Quando è presente un’operazione in cui uno dei due operandi è un `float`, anche l’altro viene convertito in `float` (nello stesso modo se sono coinvolti due operandi interi di lunghezza diversa, il tipo più piccolo viene convertito nel tipo più grande).
Nel primo caso per eseguire la moltiplicazione tra 5 e (`far-32`), il 5 viene convertito in `float` (così come il 32 prima di eseguire la sottrazione) ed il risultato viene poi diviso per 9, anch’esso convertito prima in `float`. Complessivamente viene ottenuto un numero `float` assegnato a `cel`.
Nel secondo caso invece viene prima effettuata una divisione tra numeri interi (5/9), che dà come risultato intero 0, che poi viene convertito in `float` e moltiplicato per (`far-32`). Il risultato chiaramente è sempre 0!
Questo esempio fa capire come in molti casi la conoscenza dei meccanismi interni del compilatore può essere utile per prevenire errori. Va comunque sottolineato che questo è un caso limite, che si verifica soprattutto usando la divisione con operandi interi ed in pochi altri casi.

3. Le istruzioni condizionali

INTRODUZIONE

Con il C è possibile scrivere programmi di tipo procedurale ovvero elenchi di istruzioni da eseguire in sequenza che compongono, appunto, una procedura. Durante l'esecuzione di una procedura, è spesso necessario dover scegliere tra diverse possibili azioni sulla base di un determinato evento o stato di funzionamento. In altri casi è invece necessario che una stessa sequenza di operazioni possa essere ripetuta più volte, fino al raggiungimento di un obiettivo prefissato. Il C mette a disposizione del programmatore una serie di costrutti in grado di deviare il normale flusso sequenziale di esecuzione delle istruzioni, al verificarsi di particolari condizioni. Questi costrutti sono normalmente divisi in due sezioni: una in cui è valutata la condizione, l'altra in cui sono elencate le istruzioni da eseguire. In questo capitolo verranno illustrate le istruzioni condizionali previste dal linguaggio C ed il loro utilizzo nei programmi.

L'ISTRUZIONE IF

L'istruzione `if` (dall'inglese "se") è la più classica delle istruzioni condizionali ed esiste in pratica in qualsiasi linguaggio di programmazione. La sua sintassi è il seguente:

```
if (condizione) {
  …elenco delle
  istruzioni da eseguire
  se la condizione è
  vera…
}
```

Se la condizione espressa all'interno delle parentesi tonde risulta vera, viene eseguito il blocco di istruzioni inserite tra le parentesi graffe, in caso contrario, il blocco tra le parentesi graffe viene ignorato ed il programma riprende l'esecuzione dall'istruzione immediatamente successiva alle parentesi. In C, la condizione vero/falso è rappresentata numericamente nel seguente modo: è falso un numero che vale zero ed è vero un numero diverso da zero.
Si consideri l'esempio seguente:

```
main() {
  int a;

  a=1;
  if (a) {
    printf("Condizione vera\n");
  }
}
```

Compilando ed eseguendo in programma, verrà stampato a video il testo “Condizione vera”.
Si modifichi ora il codice assegnando alla variabile `a` il valore 0:

```
main() {
  int a;

  a=0;
  if (a) {
    printf("Condizione vera\n");
  }
}
```

In questo caso non viene stampato alcun testo, a conferma che il codice all'interno della `if` non è stato eseguito. Assegnando alla variabile `a` uno dei seguenti valori -1, 10 o -500, otterremo di nuovo il messaggio “Condizione vera”.
Si consideri ora il codice modificato come segue:

```
main() {
  int a;

  a=1;
  if (a) {
    printf("Condizione vera\n");
  } else {
    printf("Condizione falsa\n");
  }
}
```

In questo modo “se” la condizione è vera, viene visualizzata la frase “Condizione vera”, altrimenti (in inglese “else”) viene visualizzata la frase “Condizione falsa”.
Si provi ora ad inserire al posto della variabile una condizione da valutare:

```
main() {
  int a;

  a=50;

  if (a==50) {
    printf("A uguale a 50\n");
  };
```

```
  if (a<50) {
    printf("A minore di 50\n");
  };

  if (a>50) {
    printf("A maggiore di 50\n");
  };
}
```

In questo caso è stato usato un diverso costrutto `if` per ciascuna delle condizioni da testare. Nel primo viene analizzata la condizione (`a==50`) ovvero se `a` è uguale al valore 50, nel secondo se `a` è minore di 50 ed infine nel terzo se `a` è maggiore di 50. Da notare come sia diverso l'operatore da usare nel caso della comparazione tra due valori da quello di assegnazione di un valore ad una variabile.
La condizione `a==50` significa: “confronta il valore contenuto nella variabile `a` con la costante numerica 50”. Il risultato di questo confronto può essere vero o falso. Falso, se la variabile `a` contiene un valore diverso da 50. Vero se `a` vale proprio 50. Notare che la notazione `a=50` invece significa: “inserisci nella variabile `a` il valore 50”. Il risultato di questa operazione è sempre vero.
All'interno delle parentesi tonde può essere inserita una qualsiasi espressione. Per decidere se eseguire o no il blocco di istruzioni seguente viene sempre e solo utilizzato il risultato dell'espressione.
Ecco un altro esempio:

```
main() {
  int a;

  a=1;
  if ((a%2)==1) {
    printf("Condizione vera\n");
  } else {
    printf("Condizione falsa\n");
  }
}
```

Il comportamento di questo programma non cambia rispetto agli esempi precedenti. È sempre il valore `a` che determina se il risultato dell'espressione è vero o falso.
Se l'istruzione da eseguire a condizione soddisfatta è una sola, è possibile omettere le parentesi graffe ed alleggerire il codice sorgente:

```
main() {
  int a;

  a=1;
  if (a)
    printf("Condizione vera\n");
  else
    printf("Condizione falsa\n");
}
```

L'ISTRUZIONE SWITCH

Come l'`if` anche l'istruzione `switch` consente di includere o escludere porzioni di programma in base al valore assunto da una determinata espressione. Contrariamente alla `if`, che consente di distinguere solo se una condizione è vera o falsa, la `switch` può distinguere qualsiasi possibile valore e determinare l'esecuzione di determinate parti di codice. Di seguito un esempio:

```
main() {
   int a;

   a=0;

   switch(a) {
    case 0:
      printf("A uguale a zero\n");
    break;

    case 1:
      printf("A uguale a uno\n");
    break;

    case 2:
      printf("A uguale a due\n");
    break;

    default:
      printf("A diverso da 0, 1 e 2\n");
    break;
   }
}
```

Come per la `if` anche nella `switch` la condizione è espressa all'interno delle parentesi tonde. All'interno delle parentesi graffe è invece possibile inserire più blocchi di istruzioni, delimitati dalle parole `case` e `break`. Ogni sezione `case` identifica la porzione di codice da eseguire per uno specifico valore. Se viene omessa l'istruzione `break`, anziché uscire dalla condizione, verrà eseguito il `case` successivo. Se il valore della condizione di `switch` non coincide con nessuno dei valori della case allora viene eseguito il blocco identificato dalla parola `default`. Purtroppo, in ogni `case`, è possibile solo specificare un unica costante numerica, per cui non è possibile comparare la condizione di `switch` con un intervallo di valori o il contenuto di una variabile.

IL CICLO FOR

L'istruzione `for` permette di creare dei cicli (*loop*), cioè di fare in modo che una sezione del programma sia ripetuta un certo numero di volte. La sintassi dell'istruzione `for` è la seguente:

```
for (condizione iniziale;
  condizione di permanenza;
  espressione di iterazione)
{
  …elenco delle
  istruzioni da eseguire
  per ogni ciclo…
}
```

La condizione iniziale è un'espressione opzionale che viene eseguita prima di iniziare il ciclo, tipicamente contiene l'inizializzazione della variabile utilizzata come contatore. La condizione di permanenza è un'espressione che, quando diviene falsa, causa l'interruzione del ciclo. L'espressione di iterazione è eseguita alla fine di ogni ciclo ed è utilizzata tipicamente per aggiornare la variabile di conteggio.
Ad esempio, per contare da 1 a 10, può essere usato il seguente programma:

```
main() {
  int i;

  for(i=1; i<=10; i++) {
    printf("Conto %d\n",i);
  }

  printf("Fatto!\n");
}
```

La prima espressione tra le parentesi tonde è `i=1` e viene eseguita solo all'inizio del loop. Le due espressioni successive vengono valutate dopo aver eseguito tutte le istruzioni tra le parentesi graffe. L'espressione `i<=10` è la condizione di permanenza nel ciclo. In altre parole il ciclo verrà ripetuto fino a che questa espressione rimane vera (cioè fino a che la variabile `i` è minore o uguale a 10). L'espressione `i++` è invece eseguita a fine ciclo, per incrementare di uno la variabile di conteggio `i`.
Tutte le tre espressioni all'interno delle parentesi tonde dell'istruzione `for` possono essere omesse.
Si consideri infatti il seguente programma:

```
main() {
  int i;

  for(i=1; ; i++) {
    printf("Conto %d\n",i);
  }

  printf("Fatto!\n");
}
```

In questo caso il conteggio prosegue all'infinito poiché manca la condizione di arresto. Per terminare l'esecuzione del programma, sarà necessario premere i tasti CTRL + C (su PC).
Un loop infinito può essere ottenuto più semplicemente omettendo tutte le condizioni:

```
main() {
  for(;;)
    printf("Ciao\n");
}
```

È possibile influenzare l'esecuzione del ciclo `for` anche all'interno delle parentesi graffe. Per far questo esistono due istruzioni: `break` per interrompere immediatamente l'esecuzione del ciclo e `continue` per passare alla successiva iterazione senza completare il blocco di istruzioni. Ad esempio:

```
main() {
  int i;

  for(i=1; ; i++) {
    printf("Conto %d\n",i);
    if (i<10) continue;
    printf("Fatto!\n");
    break;
  }
}
```

Questo programma effettua ancora un conteggio fino a 10, ma con alcune varianti che evidenziano il ruolo delle istruzioni `break` e `continue`. Da notare, anzitutto, che è stata omessa l'espressione i<=10 all'interno delle parentesi tonde, per ottenere un ciclo infinito. Dopo la scrittura del valore di `i` a video è stata inserita una `if`:

```
if (i<10) continue;
```

Questa riga istruisce il microprocessore a continuare con la prossima iterazione del ciclo, senza eseguire le istruzioni presenti alle righe successive. Il controllo passa quindi sempre all'inizio del ciclo finché i risulta minore di 10. Quando `i` vale 10 verranno finalmente eseguite le due righe sottostanti e quindi l'istruzione `break`, che determina l'uscita dal ciclo.

IL CICLO WHILE

L'istruzione `while` è molto simile alla for, infatti permette di ripetere una sezione di codice finché (`while` significa proprio "*mentre*" in inglese) è verificata una certa condizione. Un ciclo realizzato con l'istruzione `while` ha la seguente forma:

```
while (condizione di permanenza)
{
  …elenco delle
  istruzioni da eseguire
  per ogni ciclo…
}
```

In pratica l'istruzione `while` equivale ad un'istruzione `for` in cui è specificata solo la

condizione di arresto:

```
for (;condizione di permanenza;) {
   ... elenco delle
   istruzioni da eseguire
   per ogni ciclo
}
```

Il conteggio da 1 a 10 visto in precedenza può essere riscritto utilizzando l'istruzione `while` nel seguente modo:

```
main() {
   int i;

   i=1;
   while(i<=10) {
      printf("Conto %d\n",i);
      i++;
   }

   printf("Fatto!\n");
}
```

In questo caso è stato inizializzato il valore del contatore (`i=1`) prima del ciclo stesso, incrementandolo (`i++`) alla fine del blocco tra parentesi. Il risultato ottenuto è identico a quello ottenuto con l'istruzione `for`.
La scelta tra `while` e `for` dipende ovviamente dal contesto e può contribuire a scrivere sorgenti più facilmente leggibili. L'istruzione `while` è usata frequentemente, soprattutto nel campo dei sistemi embedded, per implementare cicli infiniti:

```
main() {
   while(1) {
   …
   };
}
```

Dal momento che "1" è sempre vero, il codice tra parentesi sarà reiterato all'infinito.

IL COSTRUTTO DO… WHILE

Una forma di ciclo leggermente diversa è invece rappresentata dalle istruzioni `do` e `while` utilizzate congiuntamente. Si consideri nuovamente l'esempio del conteggio da 1 a 10, modificato con l'uso del costrutto `do…while`:

```
main() {
   int i;
   i=1;
```

```
  do {
    printf("Conto %d\n",i);
    i++;
  } while (i<=10);

  printf("Fatto!\n");
}
```

La condizione di permanenza nel ciclo (`i<=10`) viene posta al termine del ciclo stesso e questo garantisce che il codice venga eseguito almeno una volta, prima di testare la condizione (ed eventualmente uscire).

ALTRI MODI PER DEVIARE IL FLUSSO DI ESECUZIONE

Per deviare il flusso di un programma esistono ancora altri modi. Un modo molto semplice è rappresentato dall'istruzione `goto` (dall'inglese "vai a"). Usando questa istruzione è possibile saltare in un punto qualsiasi del programma, a cui era stato preventivamente assegnata una "etichetta" (`label`). Di seguito un esempio di uso dell'istruzione `goto` nel programma per il conteggio da 1 a 10:

```
main() {
  int i;
  i=1;

InizioCiclo:

  printf("Conto %d\n",i);
  i++;
  if (i<=10) goto InizioCiclo;
  printf("Fatto!\n");
}
```

La stringa "`InizioCiclo:`" è una label ovvero un riferimento assoluto ad un punto preciso del programma. È possibile scegliere dei nomi arbitrari per le label, purchè siano seguiti dai due punti e che non siano una delle parole riservate del C (ad esempio `for`, `while`, `main`, etc.). L'esecuzione di `goto InizioCiclo` costringe il microprocessore a saltare nel punto di programma dove è stata inserita l'etichetta `InizioCiclo`.
L'istruzione `goto` utilizzata singolarmente non ha una grande utilità (anzi esistono di solito delle alternative più eleganti e funzionali), un uso intensivo di questa istruzione può rendere il codice poco leggibile. Per questo motivo l'istruzione `goto` è utilizzata raramente dalla maggior parte dei programmatori. Tuttavia l'uso della `goto` in certi contesti (non molto frequenti) può risultare comodo e può portare perfino ad una semplificazione o ad una più spinta ottimizzazione del codice.

4. Primi esempi di codice

INTRODUZIONE

Per comprendere meglio quanto trattato negli scorsi capitoli e per acquisire maggiore confidenza con gli elementi del linguaggio, verranno qui mostrati una serie di programmi di esempio più complessi e per ciascuno di essi verrà fornito un commento del codice ed una spiegazione dettagliata del funzionamento. I programmi presentati sono comunque molto semplici, tuttavia la loro struttura e gli accorgimenti utilizzati hanno un notevole valore didattico, che fornirà le basi per la comprensione degli argomenti trattati nei successivi capitoli.

CALCOLO DEL FATTORIALE

Il seguente programma mostra uno dei possibili metodi per il calcolo del fattoriale di un numero naturale (cioè di un intero positivo maggiore o uguale ad 1). Per chiarezza è il caso di ricordare che il fattoriale di un numero naturale N (indicato con "N!") è definito come il prodotto dei primi N numeri naturali: ad esempio il fattoriale di 3 è 1x2x3 = 6. Una caratteristica interessante del fattoriale è che assume valori grandissimi anche per valori di N abbastanza piccoli. Un esempio di codice per il calcolo del fattoriale è il seguente:

```
#include <stdio.h>

main() {
  int i, n;
  float fatt=1;

  printf("Fattoriale di: ");
  scanf("%d", &n);

  for(i=1; i<=n; i++)
    fatt = fatt*i;

  printf("%d! = %f", n, fatt);

}
```

Nel programma vengono dichiarate due variabili intere (`i` ed `n`), utilizzate per eseguire il conteggio ed una variabile di tipo `float`, utilizzata per il calcolo del fattoriale vero e proprio.
Pur essendo coinvolti solo numeri interi non è conveniente utilizzare un tipo intero per la variabile `fatt` in quanto il risultato del fattoriale sarà talmente grande da non essere rappresentabile con un tipo `int` o `unsigned long`. Un `unsigned long` infatti potrebbe contenere al massimo il fattoriale di 12 e per valori maggiori i risultati sarebbero inconsistenti. È comunque istruttivo provare a sostituire il `float` con un `unsigned long` (quindi anche `%f` con `%d` nella `printf`).
A parte queste considerazioni "di progetto", il resto del programma è abbastanza semplice: viene richiesto all'utente il valore di cui calcolare il fattoriale, utilizzando una `scanf` e successivamente il fattoriale vero e proprio è calcolato come da definizione, moltiplicando i primi N numeri naturali con un ciclo `for`.
Notare come la variabile `fatt` sia utilizzata da "accumulatore" nel calcolo. Notare anche che, sarebbe stato possibile evitare di utilizzare la variabile `i`, sfruttando meglio `n` nel ciclo `for` o utilizzando l'istruzione `while` al suo posto. Si consiglia di provare a modificare in tal senso il programma come esercizio.

CALCOLO DELLA MEDIA

Si supponga di avere una serie di dati numerici e di volerne calcolare la media aritmetica. Il programma seguente richiede il numero dei dati di cui si vuole calcolare la media, il valore di ognuno di essi, per poi effettuare il calcolo della media sommando i singoli dati e dividendo per il numero dei dati stessi.

```
#include <stdio.h>

main() {
   int i, n;
   float dato, med;

   printf("Numero dati: ");
   scanf("%d", &n);

   for(i=0; i<n; i++) {
      printf("Dato n. %d: ", i+1);
      scanf("%f", &dato);
      med = med + dato;
   }

   med = med / n;

   printf("Media = %.2f", med);

}
```

Anche in questo caso il programma è piuttosto semplice da comprendere. La variabile `n` viene utilizzata per tenere conto del numero di elementi da inserire, mentre i valori immessi

ed il calcolo parziale vene eseguito con valori `float`. Il risultato parziale viene dapprima accumulato nella variabile `med` e solo alla fine del ciclo viene eseguita la divisione per il numero di dati immessi. Da notare come nella stringa della funzione `printf` sia stato utilizzato il carattere di conversione "`%.2f`" anziché il semplice "`%f`". Il valore specificato indica il numero di decimali da visualizzare dopo la virgola ed è uno dei tanti accorgimenti messi a disposizione dalla `printf` per "formattare" il testo visualizzato.
Anche in questo caso è utile provare a sostituire i `float` con degli interi.

INDOVINA IL NUMERO

Il programma che segue implementa un semplice gioco e costituisce un esempio piuttosto realistico di quello che può essere il controllo di flusso di un programma in una reale applicazione. L'utente deve indovinare un numero compreso tra 1 e 10 scelto a caso dal programma. Sono consentiti solo 6 tentativi e ad ogni risposta dell'utente il programma fornisce un'indicazione di aiuto.

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

main() {
  int numero,
      tentativi,
      risposta;

  tentativi=6;
  /* genera numero casuale
     compreso tra 1 e 10 */
  numero = rand()%10 + 1;

  do {
    printf("Indovina il numero: ");
    scanf("%d", &risposta);

    if (risposta==numero) {
      /* Risposta giusta! */
      printf("*** Indovinato! ***\n");
      break;
    }
    else {
      /* Risposta sbagliata! */
      if (risposta<numero)
        printf("troppo piccolo...\n");
      else
        printf("troppo grande...\n");
    }

    tentativi--;
```

```
    } while (tentativi > 0);

}
```

All'inizio del codice oltre alla *stdio.h* è stata inclusa anche un'altra libreria: *stdlib.h*. Si tratta di un'altra libreria standard dell'ANSI C che contiene una funzione per la generazioni di numeri casuali. Le funzioni delle principali librerie standard verranno analizzate meglio in uno dei prossimi capitoli.
Le variabili dichiarate sono soltanto tre: una viene utilizzata per memorizzare il numero casuale generato dal programma, un'altra per tenere conto del numero di tentativi rimanenti e l'ultima per memorizzare il valore fornito dall'utente. Dal momento che tutti i numeri coinvolti nel programma sono sufficientemente piccoli (teoricamente compresi tra 0 e 10) sarebbe stato possibile utilizzare il tipo `char` per tutte le tre variabili, ottenendo così un certo risparmio di memoria. È possibile modificare il programma in tal senso, ponendo attenzione alla variabile `risposta`, che deve essere necessariamente un `int`, dal momento che la funzione `scanf` si aspetta una variabile di questo tipo.
Il numero casuale è generato grazie alla funzione `rand()` della libreria *stdlib.h*, che restituisce un numero casuale intero positivo compreso tra 0 e 2^{32} (quest'ultimo valore dipende dal compilatore utilizzato). Per ottenere un numero compreso tra 1 e 10, viene utilizzato l'operatore % (modulo o resto della divisione), ottenendo un valore compreso tra 0 e 9 e sommato a questo 1.
Scelto il numero casuale, viene avviato il ciclo in cui si chiede all'utente di indovinarlo. Il ciclo è implementato con un costrutto `do…while`, che esegue il codice al massimo 6 volte (numero dei tentativi a disposizione). All'interno del loop viene chiesto all'utente un numero, successivamente verificato se esso coincide col numero casuale scelto prima. In caso affermativo viene stampata una stringa di congratulazioni e viene abbandonato il loop grazie all'istruzione `break`. Se il numero non è quello corretto, viene eseguito un controllo per stabilire se è più piccolo o più grande di quello da indovinare e stampata la stringa corrispondente. Alla fine del loop viene decrementato il numero di tentativi disponibili.

SEMPLICE CALCOLATRICE

L'esempio seguente mostra come il costrutto `switch` può essere usato per implementare una semplice calcolatrice dotata delle operazioni aritmetiche di base. Il programma riesce ad interpretare semplici espressioni a due operandi del tipo "3 + 9" o "286.5 / 12e-2" e fornisce il risultato del calcolo.

```
#include <stdio.h>

main() {
  char sel;
  float op1, op2, res;

  /* stampa titolo */
  printf("*** Calcolatrice ***\n\n");

  /* ciclo principale */
```

```
  while(1) {
    printf("Calcola: ");
    scanf("%f %c %f", &op1, &sel, &op2);

    switch(sel) {

      /* somma */
      case '+':
       res = op1 + op2;
      break;

      /* differenza */
      case '-':
       res = op1 - op2;
      break;

      /* prodotto */
      case '*':
       res = op1 * op2;
      break;

      /* quoziente */
      case '/':
       res = op1 / op2;
      break;

      /* non riconosciuto */
      default:
       printf("Errore di sintassi!\n");
       res = 0;
      break;
    }
    printf("Risultato: %f\n\n", res);
  }
}
```

In questo caso vengono utilizzate tre variabili di tipo `float` per contenere i due operandi ed il risultato ed una variabile di tipo `char` per il tipo di operazione. Il corpo principale del programma è costituito da un loop infinito ottenuto con l'istruzione `while(1)`. Inizialmente viene richiesto all'utente di immettere l'espressione da valutare, utilizzando l'istruzione `scanf`. Va notato che in generale la decodifica di questo tipo di input non è per niente semplice e richiede un codice abbastanza complesso per eseguire tutti i controlli necessari a verificare che non siano stati commessi degli errori e per ricavare dalla stringa tutte le informazioni necessarie in termini di operatori, operandi e segni particolari (queste operazioni vengono chiamate "*parsing*"). In questo è stato supporto che il nostro input abbia un formato definito abbastanza rigidamente, sia come contenuto, sia come formattazione. In questi casi è possibile sfruttare le capacità della funzione `scanf` di ricavare più campi da un'unica stringa. Per fare questo è sufficiente specificare i diversi tipi di dati nella stringa di formattazione ed elencare di seguito le rispettive variabili che li dovranno contenere.

Sia i caratteri di conversione che le variabili devono essere elencate nello stesso ordine in cui compariranno nella stringa di input. Nel caso trattato, come già detto, la stringa deve essere scomposta in tre campi: il primo operando (di tipo `float`), l'operatore aritmetico (di tipo `char`) ed il secondo operando (sempre di tipo `float`). Per fare questo è stata utilizzata la stringa di formattazione "`%f %c %f`". L'uso del tipo `char` per individuare l'operatore aritmetico forse merita una spiegazione aggiuntiva. La funzione `scanf` restituirà nella variabile `sel` il codice ASCII del carattere utilizzato tra i due numeri, questo sarà un numero compreso tra 0 e 127 ed in particolare: 43 per il "+", 45 per il "-", 42 per il "*" e 47 per il "/". Sarà quindi possibile utilizzare questi numeri per decidere quale operazione eseguire. In particolare questa selezione viene effettuata utilizzando il costrutto `switch – case`, in cui nei vari `case` sono utilizzati proprio i valori corrispondenti agli operatori riconosciuti.
Quando vengono utilizzati dei caratteri non è necessario conoscerne il codice ASCII, infatti il linguaggio C permette di utilizzare direttamente i caratteri per indicare il loro valore. Questo è quello che viene fatto nei vari case del programma, in cui invece di utilizzare un valore numerico sono stati utilizzati i caratteri stessi, racchiusi tra apici (nel caso specifico: '+', '-', '*', '/'). Scrivere un carattere tra apici in C equivale a scrivere il valore del codice ASCII e questo vale anche per le stringhe di formattazione utilizzate nella funzione `printf`, ad esempio '`\n`' (a capo, codice ASCII 10), '`\t`' (tabulazione, codice ASCII 9)…
Una volta selezionato il `case` relativo all'operatore, viene eseguito il calcolo corrispondente, il cui risultato verrà poi stampato. Se per errore viene utilizzato un carattere non riconosciuto, viene eseguito il caso di `default`, che semplicemente segnala questa condizione anomala.
Un'ultima nota sulla funzione `scanf`: specificando un certo numero di campi da recuperare, la funzione "pretenderà" di leggerli e non permetterà di continuare l'esecuzione del programma fino a quando non avrà ottenuto tutti i valori. Questo significa che se viene indicata una stringa con un formato diverso da quello atteso, possono verificarsi degli errori o il programma può bloccarsi o terminare in maniera inattesa. Inoltre, a seconda delle implementazioni (cioè dal compilatore utilizzato), la funzione `scanf` può richiedere o meno uno spazio tra i vari campi.

CONVERSIONE IN BINARIO ED ESADECIMALE

Il programma seguente realizza la conversione di un numero fornito dall'utente in esadecimale ed in binario, visualizzando il risultato sullo schermo. L'esempio risulta utile per precisare alcuni particolari che hanno a che fare proprio con la conversione e l'uso di numeri in diverse basi da quella decimale.

```
#include <stdio.h>

main() {
  unsigned long n;
  char i;

  printf("Numero decimale: ");
  scanf("%D", &n);

  printf("Valore hex: %08X\n", n);
```

```
    printf("Valore binario [LSB...MSB]: ");
    for(i=0; i<32; i++) {
      if ((n%2)==0)
        printf("0");
      else
        printf("1");
      n=n/2;
    }
  }
```

Il numero immesso dall'utente viene memorizzato in una variabile di tipo `unisgned long` (variabile a 32 bit). Notare il carattere "`%D`" (maiuscolo) utilizzato nella `scanf` proprio per sottolineare la presenza di un valore di tipo `long`.
La conversione in esadecimale viene eseguita direttamente dalla `printf`. Per fare questo è sufficiente utilizzare il carattere di conversione "`%X`". Lo "08" che è stato aggiunto serve per specificare il numero di cifre esadecimali da utilizzare per rappresentare il numero (in questo caso 8 cifre con zeri iniziali sempre visibili).
Il carattere di conversione "`%X`" (o anche "`%x`") può essere utilizzato anche per la `scanf`, quando sia prevista l'immissione di valori esadecimali. A tal proposito è il caso di ricordare che in C è possibile esprimere anche nel codice i numeri sia in formato decimale, che esadecimale o ottale, indifferentemente. Per esprimere un numero in esadecimale basta anteporre il suffisso "`0x`", mentre per esprimerlo in ottale basta anteporre uno 0 (e ovviamente utilizzare cifre minori di 8). Ad esempio "`0x1C`" (esadecimale) equivale a 28 decimale, mentre "013" (ottale) a 11.
Per convertire il numero in binario nel programma è stata utilizzata una tecnica poco efficiente ma abbastanza intuitiva: viene verificato se il numero è pari o dispari controllando se il resto della sua divisione per 2 è 0 oppure 1 e scritto a schermo la cifra corrispondente. Successivamente viene diviso per due, quindi ripetuto il procedimento. Questi due passi equivalgono rispettivamente a testare il bit meno significativo ed a scorrere a destra il numero di un bit. Il numero binario ottenuto a schermo è sempre esatto (anche nel caso che venga fornito dall'utente un numero negativo), però è visualizzato a partire dal bit meno significativo al più significativo (di solito si usa la rappresentazione opposta).

5. Le funzioni ed il passaggio di variabili

INTRODUZIONE

Scrivendo un programma può capitare di dovere utilizzare in più punti frammenti di codice quasi identici, che svolgono esattamente le stesse funzioni (a meno del valore di qualche parametro). In questi casi risulta utile potere scrivere il codice in questione una sola volta e richiamarlo in tutti quei punti in cui è necessario utilizzarlo. Questo è esattamente il concetto di *funzione* in C (ed anche in altri linguaggi ad alto livello). Una funzione è proprio un frammento di codice, a cui viene dato un nome, che può essere richiamato da diversi punti del codice, eventualmente passando alcuni parametri e che esegue delle elaborazioni, eventualmente restituendo un risultato. Nel capitolo 3 è stato visto come poter deviare il flusso di esecuzione di un programma utilizzando gli operatori condizionali. Si consideri l'esempio con l'operatore `for`:

```
main() {
  int i;
  for(i=1;i<=10;i++) {
    printf("Conto %d\n",i);
  }
  printf("Fatto!\n");
}
```

Per rendere più leggibile e versatile questo programma è possibile utilizzare una funzione come segue:

```
main() {
  conta_fino_a(10);
}
```

In questo caso è stata creata una funzione a cui è stato assegnato (arbitrariamente) il nome `conta_fino_a`. A questa funzione viene passato un valore tra parentesi che rappresenta il valore finale del conteggio. Una volta definita la funzione, essa può essere richiamata quante volte si vuole all'interno di un programma. Il codice della funzione è descritto nello stesso programma, fuori dal main (che a sua volta, lo ricordiamo, è una funzione), nel modo seguente:

```
conta_fino_a(int valore) {
  int i;
```

```
  for(i=1;i<=valore;i++) {
    printf("Conto %d\n",i);
  }
  printf("Fatto!\n");
}
```

Come accennato anche nei primi capitoli, la funzione è costituita da un'intestazione (il suo nome), dalle variabili passate come argomenti e dal codice vero e proprio che, in questo esempio, è l'intero codice utilizzato negli esempi relativi all'istruzione `for`. In questo caso la funzione accetta come parametro un numero intero contenuto nella variabile `valore`. È stata quindi ottenuta una funzione di conteggio che consente di concentrare in una unica sezione del programma una determinata funzione logica consentendo di alleggerire la porzione principale del programma. La scelta del nome da adottare per le funzioni è molto importante al fine di scrivere del codice auto-descrittivo, facilmente leggibile e comprensibile. La funzione `conta_fino_a(10)` non ha bisogno di spiegazioni particolari per fare capire quale sia il suo compito. Utilizzando questa convenzione il programma assumerà sempre più l'aspetto di una descrizione in linguaggio naturale di quello che dovrà fare il microprocessore. L'uso delle funzioni in un programma ha anche un ulteriore vantaggio: permette di progettare più facilmente il programma stesso. È possibile infatti seguire un approccio *top-down*, cioè scomporre inizialmente il programma in blocchi logici più astratti, quindi dettagliare via via il loro contenuto a livello di implementazione.

VISIBILITÀ DELLE VARIABILI

All'interno delle funzioni è possibile dichiarare delle nuove variabili, che potranno essere utilizzate all'interno della funzione, ma verranno distrutte quando la funzione terminerà. Non solo, le variabili dichiarate dentro ogni funzione possono anche avere lo stesso nome di altre variabili dichiarate in altre funzioni (compreso il `main`), ma non interferiscono in nessun modo tra di loro. Per mostrare quanto detto si consideri il seguante programma:

```
main() {
  int a;
  int b;
  a=1;
  b=2;
  printf("a (main) = %d\n",a);
  printf("b (main) = %d\n",b);
  funzione();
  printf("a (main) = %d\n",a);
  printf("b (main) = %d\n",b);
}

funzione () {
  int a=3;
  int b=4;
  printf("a (funzione) = %d\n",a);
  printf("b (funzione) = %d\n",b);
}
```

Il risultato è:

```
a (main) = 1
b (main) = 2
a (funzione) = 3
b (funzione) = 4
a (main) = 1
b (main) = 2
```

Anche se nella funzione `funzione` sono state dichiarate due variabili con lo stesso nome di quelle utilizzate nel `main` ed a queste è stato assegnato un valore diverso, non si sono avute modifiche nelle prime. Questo tipo di variabili si chiamano variabili locali ovvero variabili esistenti e visibili solo all'interno della funzione. Per capire questo meccanismo è bene sapere che il C memorizza le variabili nello *stack* ovvero in una particolare area di memoria paragonabile ad una catasta in cui tutto quello che viene aggiunto per ultimo deve essere tolto per primo. Specificando una variabile con un costrutto del tipo:

```
int i;
```

si aggiunge alla catasta uno spazio di memoria in grado di contenere un valore numerico. Lo stack viene utilizzato anche per memorizzare il punto del programma in cui è stata chiamata una funzione per poter riprendere l'esecuzione dall'istruzione successiva al termine dell'esecuzione della funzione stessa. Se le righe di programma fossero numerate nel modo seguente:

```
01 main() {
02   conta_fino_a(10);
03 }

04 conta_fino_a(int valore) {
05   int i;
06
07   for(i=1;i<=valore; i++) {
08     printf("Conto %d\n",i);
09   }
10   printf("Fatto!\n");
11 }
```

Il microprocessore inizia l'esecuzione dalla linea 01, per poi passare alla linea 02, quindi salta all'istruzione alla riga 04 per arrivare fino alla riga 11, dopodichè riprenderà dalla linea 03 per poi uscire. In fase di compilazione il nome di funzione `conta_fino_a` viene tradotto nell'indirizzo di memoria corrispondente dove si trova esattamente memorizzata la funzione, per cui in andata è facile per la CPU saltare alla giusta locazione dove poter trovare la funzione da eseguire. Prima di saltare però deve memorizzare nello stack il valore di ritorno (in questo caso 03) in modo da poter tornare nel punto giusto una volta terminata l'esecuzione della funzione. Una volta entrato nella funzione `conta_fino_a` lo stack conterrà l'indirizzo di ritorno 03.
Quando incontra la definizione "`int i`" lo spazio per contenere la variabile "`i`" viene riservato nello stack, per cui lo stack conterrà la variabile "`i`" sovrapposta all'indirizzo 03. Una volta terminata l'esecuzione la CPU, per poter recuperare l'indirizzo di ritorno

è costretta a togliere dallo stack tutte le variabili sovrapposte all'indirizzo di ritorno che verranno quindi perse completamente.
In molti casi è comodo creare delle variabili visibili e modificabili da tutte le funzioni dichiarandole fuori dal corpo delle funzioni. Queste variabili vengono definite *globali*:

```
#include <stdio.h>

// Variabili globali
int a;
int b;

main() {
  a=1;
  b=2;
  printf("a (main) = %d\n",a);
  printf("b (main) = %d\n",b);
  funzione();
  printf("a (main) = %d\n",a);
  printf("b (main) = %d\n",b);
}

funzione() {
  a=3;
  b=4;
  printf("b (funzione) = %d\n",a);
  printf("b (funzione) = %d\n",b);
}
```

Il cui risultato è il seguente:

```
a (main) = 1
b (main) = 2
a (funzione) = 3
b (funzione) = 4
a (main) = 3
b (main) = 4
```

In questo esempio le due variabili visibili all'interno di main sono fisicamente le stesse visibili all'interno della funzione funzione. La modifica effettuata dentro questa funzione ha effetto sul resto del codice.

PASSAGGIO PER VALORE

L'uso di variabili globali è sempre sconsigliabile in quanto spesso fonte di errori di programmazione molto difficili da scovare in programmi complessi. È meglio quindi passare le variabili alle funzioni come argomento.
Ad esempio:

```
main() {
  int a;
  int b;
  a=1;
  b=2;
  funzione(a,b);
  printf("a (main) = %d\n",a);
  printf("b (main) = %d\n",b);
}

funzione(int a, int b) {
  a=a+3;
  b=b+4;
  printf("a (funzione) = %d\n",a);
  printf("b (funzione) = %d\n",b);
}
```

In questo caso alla funzione sono state passate le variabili dichiarate nel `main` come argomento. Tuttavia l'output del programma è il seguente:

```
a (funzione) = 4
b (funzione) = 6
a (main) = 1
b (main) = 2
```

Da questo risultato si nota che la funzione ha ricevuto il valore delle variabili dichiarate nel `main`, sebbene queste sono state modificate, non si è avuto alcun effetto fuori dalla funzione. Questo modo di passare le variabili ad una funzione si chiama *passaggio per valore*. Quello che succede è che di fatto vengono create due variabili locali dentro la funzione ed a queste viene copiato il valore di quelle passate come argomento, le variabili restano però fisicamente separate.

PASSAGGIO PER RIFERIMENTO

È possibile passare ad una funzione non il solo valore di una variabile ma la variabile vera e propria (o meglio l'indirizzo di memoria in cui è contenuta). In questo caso se la funzione modifica il valore questo verrà riportato anche nel resto del programma. Questa modalità si chiama *passaggio per riferimento* ed è realizzato nel seguente modo:

```
main() {
  int a;
  int b;
  a=1;
  b=2;
  funzione(&a,&b);
  printf("a (main) = %d\n",a);
  printf("b (main) = %d\n",b);
}
```

```
funzione(int *a, int *b) {
  *a=3;
  *b=4;
  printf("a (funzione) = %d\n",a);
  printf("b (funzione) = %d\n",b);
}
```

Il risultato è il seguente:

```
a (funzione) = 3
b (funzione) = 4
a (main) = 3
b (main) = 4
```

In questo caso le variabili `a` e `b` non sono globali, ma le modifiche apportate nella funzione `funzione` sono visibili anche nella `main`. Per ottenere il passaggio per riferimento occorre prendere come parametro della funzione il *puntatore* alle variabili (cioè il loro indirizzo di memoria). Questo si ottiene utilizzando l'operatore "*" nella funzione (che può essere interpretato come "contenuto della locazione") e l'operatore "&" ("indirizzo di") per ottenere l'indirizzo della variabile nella chiamata a funzione. Dal momento che viene passato l'indirizzo, se si intende modificare il contenuto delle variabili all'interno della funzione, è necessario usare l'operatore * prima del nome della variabile. Similmente si userà l'operatore & per passare l'indirizzo. All'inizio questo meccanismo può creare un po' di confusione, tuttavia va sottolineato che i puntatori sono uno strumento estremamente potente e versatile offerto dal linguaggio C, per questo motivo l'argomento verrà ripreso e trattato più dettagliatamente nel capitolo 9.

VALORE DI RITORNO DI UNA FUNZIONE

In molti casi è utile che la funzione chiamata, restituisca un valore come risultato delle elaborazioni compiute. Questo può essere ottenuto come visto prima (cioè passando una variabile per riferimento), ma più propriamente, nel caso sia richiesto il ritorno di un unico valore si può usare un metodo più semplice e diretto:

```
main() {
  int a;
  a=10;
  a=raddoppia(a);
  printf("Valore di a = %d\n",a);
}

int raddoppia(int a) {
  a = a+a;
  return a;
}
```

Utilizzando l'istruzione `return` alla fine di una funzione, è possibile far restituire ad essa un valore di ritorno. Come si può vedere il tipo di variabili restituito deve essere anche dichia-

rato nell'intestazione della funzione. In questo caso la funzione `raddoppia` viene trattata all'interno del main come una variabile numerica a sola lettura che può essere inserita anche in funzioni complesse.

LE MACRO

In C è possibile definire delle semplici funzioni anche sotto forma di *macro*. La differenza sostanziale tra una macro ed una funzione sta nel livello in cui viene trattata dal compilatore C. La macro consiste in una semplice sostituzione testuale all'interno del sorgente da parte del *preprocessore*, prima della compilazione vera e propria. Se viene definita una macro che viene richiamata dieci volte, il compilatore genererà dieci copie del codice corrispondente. La funzione invece viene generata in fase di compilazione ed ogni chiamata fa riferimento sempre allo stesso segmento di codice oggetto. Ecco un esempio:

```
#define raddoppia(valore) valore+valore

main() {
   int a;
   a=10;
   a=raddoppia(a);
   printf("main() -> a = %d\n",a);
}
```

Prima di compilare il sorgente in codice macchina, viene eseguito il *preprocessore* ovvero un programma che si occupa di effettuare delle elaborazioni testuali al codice sorgente. La direttiva `#define` è indirizzata proprio al preprocessore ed indica di sostituire l'espressione della prima stringa con l'espressione della seconda. Ovunque venga trovato nel sorgente l'espressione `raddoppia(a)` verrà sostituito con l'espressione "`a+a`". Per cui il codice che verrà realmente compilato sarà:

```
main() {
   int a;
   a=10;
   a=a+a;
   printf("main() -> a = %d\n",a);
}
```

L'uso delle macro è conveniente in tutti quei casi in cui la funzione che deve essere effettuata è molto semplice. Una funzione infatti introduce del codice aggiuntivo per la memorizzazione nello stack delle variabili e per il corretto ritorno alla funzione chiamante. Quando le istruzioni da eseguire sono minimali potrebbe essere conveniente evitare di inserire questo codice aggiuntivo e ricorrere quindi alle macro. A parte questo, le macro sono anche utili per migliorare la leggibilità del codice.

6. Gli operatori

INTRODUZIONE

Il linguaggio C mette a disposizione una serie di operatori aritmetici e logici che è possibile utilizzare con le variabili. Alcuni di questi operatori sono già stati utilizzati nei precedenti capitoli quindi la loro sintassi dovrebbe risultare familiare (anche perché corrisponde in molti casi alla normale sintassi delle espressioni aritmetiche). In questo capitolo verranno considerati la maggior parte degli operatori disponibili e per ciascuno di essi verrà fornita una descrizione ed un esempio di utilizzo.

OPERATORI ARITMETICI (+, -, *, /, %, =)

Come già visto il C supporta le comuni operazioni aritmetiche di somma (+), differenza (-), prodotto (*), quoziente (/) e modulo (%). Grazie a questi operatori è possibile scrivere espressioni aritmetiche utilizzando la normale sintassi matematica. Si ricorda che l'operatore "=" (assegnazione) non indica l'equivalenza tra l'espressione alla sua destra ed alla sua sinistra, ma l'assegnazione del valore dell'espressione di destra alla variabile di sinistra. Questo dovrebbe essere piuttosto intuitivo, ma è bene sottolinearlo per evitare confusione. Secondo quanto detto l'espressione:

```
a = b*c+d;
```

significa che alla variabile *a* verrà assegnato il valore ottenuto dal prodotto di *b* moltiplicato per *c*, a cui viene poi sommato il valore di *d*. Il linguaggio C segue l'usuale precedenza matematica tra gli operatori, quindi prima verranno eseguiti i prodotti e le divisioni, poi le somme e le differenze. È anche possibile utilizzare le parentesi (solo tonde) per raggruppare delle espressioni o dare delle precedenze:

```
a = (b+c)*(d – e);
```

in questo caso prima verranno eseguite la somma e la differenza, poi il prodotto dei due risultati parziali.
Come accennato nel capitolo 2 gli operatori aritmetici, ad esclusione del modulo, possono essere utilizzati sia con le variabili intere che con quelle floating point. Il modulo, che come

già detto restituisce il resto della divisione tra i due operandi interi, può essere utilizzato soltanto con i tipi interi.

NOTAZIONI ALTERNATIVE

Quando uno degli operandi di una espressione aritmetica è costituito dalla stessa variabile di destinazione, è possibile utilizzare una forma abbreviata di alcune espressioni. Ad esempio:

a = a + b;

può essere scritta nella forma equivalente:

a += b;

La stessa forma abbreviata può essere adottata per tutti gli altri operatori aritmetici e logici. Un caso molto frequente è quello di dovere incrementare o decrementare di 1 una determinata variabile, in questi casi si può utilizzare la scrittura semplificata:

a++;
b--;

che equivale all'espressione:

a = a + 1;
b = b – 1;

La disposizione degli operatori rispetto al nome della variabile ha un significato particolare. Scrivendo `a++` l'incremento viene effettuato dopo aver calcolato il valore dell'intera espressione. Scrivendo `++a` l'incremento viene effettuato prima di aver calcolato il valore dell'intera espressione. Ad esempio:

```
main () {
  int a;
  a=1;
  printf("a vale: %d\n", a++);
  printf("a vale: %d\n", a);
}
```

Visualizzerà a video:

```
a vale: 1
a vale: 2
```

mentre

```
main () {
  int a;
```

```
    a=1;
    printf("a vale: %d\n", ++a);
    printf("a vale: %d\n", a);
  }
```

visualizzerà:

```
  a vale: 2
  a vale: 2
```

Similmente, l'espressione:

a = c*b++;

sarà valutata col valore originale di *b*, che però alla fine dell'istruzione risulterà incrementata di uno.

OPERATORI RELAZIONALI E LOGICI

Gli operatori relazionali consentono di confrontare il valore di due operandi, forniscono un risultato di tipo vero o falso e vengono utilizzati quasi esclusivamente all'interno delle istruzioni condizionali.

Gli operatori di comparazione (==, !=)

Gli operatori di comparazione permettono di valutare se il valore dell'espressione di sinistra è uguale (==) o diverso (!=) rispetto a quello dell'espressione di destra. Entrambi restituiscono una condizione di verità (vero o falso).
Ecco un esempio di utilizzo dell'operatore ==:

```
main () {
   int a;
   a=1;
   if (a==1) {
     printf("a vale 1\n");
   } else {
     printf("a non vale 1\n");
   }
}
```

In questo esempio viene assegnato il valore 1 alla variabile `a` mediante l'espressione `a=1` quindi ne viene verificato il contenuto all'interno della `if` con l'espressione `a==1`. Dato che a vale proprio 1, allora la condizione all'interno della `if` sarà soddisfatta ed il programma eseguirà l'istruzione: `printf("a vale 1\n")`.

Altri operatori relazionali (<, >, ″ , ≥)

Questi operatori, come indica il nome stesso, vengono utilizzati per confrontare il valore di due operandi, per stabilire se il primo è minore (<), maggiore (>), minore o uguale (″), mag-

giore o uguale (>=) del secondo. Il valore restituito è sempre un valore di verità. Un esempio di utilizzo di questi operatori è il seguente:

```
main () {
  int a;
  a=5;
  if (a>3)
    printf("a vale piu' di 3\n");
}
```

In questo caso la condizione `a>3` sarà verificata quindi verrà stampato il testo. Questi operatori trovano spesso impiego, come già visto, all'interno dei cicli (`for`, `while`...), per verificare la condizione di arresto.

L'operatore AND (&&)

AND, dall'inglese "e" (congiunzione), è un operatore che verifica se le due espressioni, alla sua destra ed alla sua sinistra, sono entrambe vere, nel qual caso restituisce il valore "vero". Se anche solo una delle due è falsa, il risultato è falso. Ad esempio:

```
main () {
  int a=12, b=3;
  if ((a>5)&&(b<10)) {
    printf("Condizione vera\n");
  } else {
    printf("Condizione falsa\n");
  }
}
```

Nel codice riportato la condizione sarà verificata, perché `a` è minore di 5 e (AND) `b` è minore di 10.

L'operatore OR (||)

L'operatore OR, dall'inglese "O", "OPPURE", è un operatore che verifica se almeno una delle due espressioni è vera. Solo se entrambe sono false, il risultato sarà falso.

L'operatore NOT (!)

Il NOT, dall'inglese "NON", è un operatore che inverte il valore di un'espressione logica ovvero restituirà un valore vero se l'espressione è falsa e viceversa. Questo operatore può essere applicato ad una variabile o ad un'espressione più complessa compresa tra parentesi. In entrambi i casi il valore di verità dell'espressione sarà invertito:

```
main () {
  int a=5;
  if (!(a>3)) {
    printf("Condizione vera\n");
  } else {
    printf("Condizione falsa\n");
  }
}
```

In questo caso anche se la condizione a>3 è vera, la negazione la rende falsa quindi verrà visualizzata la seconda stringa.

GLI OPERATORI SUI BIT (BITWISE)

Questi operatori consentono di eseguire operazioni logiche direttamente sui bit delle variabili intere (non possono essere utilizzate per quelle floating point). Come già visto nel capitolo 2, le variabili intere (ad 8, 16 o 32 bit) sono rappresentate internamente in binario naturale ed in complemento a 2 se sono di tipo *signed*.
Queste variabili si prestano a rappresentare non solo numeri, ma in genere dati binari (per esempio dati presenti su porte di I/O o su bus, il contenuto di registri, campi di bit relativi a trame di particolari protocolli, etc.), è quindi utile ed importante potere operare su di essi a livello dei singoli bit.

Operatori AND, OR, XOR (&, |, ^)

Questi operatori permettono di eseguire le rispettive operazioni logiche tra due operandi, a livello dei singoli bit (indipendentemente).
Ad esempio:

```
main() {
  unsigned char a, b;

  a = 229; // in binario: 11100101
  b = 15; // in binario: 00001111

  printf("a AND b = %d\n", a&b);
  printf("a OR  b = %d\n", a|b);
  printf("a XOR b = %d\n", a^b);
}
```

il risultato sarà:

```
a AND b = 5
a OR  b = 239
a XOR b = 234
```

Rappresentando infatti i due operandi in binario si ha:

```
11100101 (229)
00001111 (15)
```

Eseguendo l'operazione di AND, il risultato sarà un 1 solo nelle posizioni in cui entrambi i bit degli operandi valgono 1:

```
11100101 & (229)
00001111 = (15)
00000101   (5)
```

nel caso dell'OR si ottiene un 1 nelle posizioni in cui almeno uno dei due bit vale 1:

```
11100101 | (229)
00001111 = (15)
11101111   (239)
```

mentre nel caso dell'XOR si ottiene un 1 nelle posizioni in cui i bit sono diversi:

```
11100101 ^ (229)
00001111 = (15)
11101010   (234)
```

Come esempio di impiego delle operazioni bitwise, si supponga di avere a disposizione una porta di I/O ad 8 bit e di dovere leggerla e scriverla nel corso delle operazioni. L'operatore AND può essere utile in due casi: in lettura quando si presenta la necessità di sapere se determinati bit valgono 1, oppure in scrittura quando devono essere azzerati soltanto alcuni bit.
Queste operazioni vengono realizzate creando una "maschera" che individua i bit che interessano, quindi eseguendo un AND con il dato originale: l'effetto sarà quello di "ritagliare" dal dato originale soltanto i bit che interessano. Per testare il quarto bit di un byte si può operare nel seguente modo:

```
11101001 & (dato originale)
00001000 = (maschera)
00001000   (dato mascherato)
```

Il dato ottenuto quindi vale 0 se il bit nella posizione evidenziata vale 0, oppure un numero diverso da 0 se in quella posizione è presente un uno. Per testare la presenza di un 1 sul quarto bit si può quindi usare il seguente codice:

```
…
dato=leggi_porta();
if (dato&0x08)
  printf("Bit 4 = 1");
…
```

Per "ritagliare" soltanto i 4 bit meno significativi ed azzerare i restanti, sarà necessario un AND con 00001111, cioè 0x0F (di solito i numeri la cui rappresentazione binaria è significativa vengono espressi in esadecimale, perché è facile convertire "a occhio" da base 16 a base 2 e viceversa). Al contrario per azzerare solo il primo bit, sarà necessario usare come maschera 0xFE (11111110). In codice:

```
dato=dato&0xFE;
// In versione compatta:
// dato&=0xFE;
```

L'operatore OR, al contrario, viene usato di solito per impostare ad 1 determinati bit. Anche in questo caso è sufficiente eseguire l'operazione di OR tra il dato ed una maschera. Ad esempio per impostare ad 1 i bit 7 ed 8 di una porta o di un dato:

```
00001001 | (dato originale)
11000000 = (maschera, 0xC0 in Hex)
11001001   (dato mascherato)
```

In codice:

```
dato=dato|0xC0;
// In versione compatta:
// dato|=0xC0;
```

L'operatore XOR viene usato di solito per negare determinati bit di un dato di partenza. Ad esempio la seguente istruzione nega il bit più significativo del dato originale:

```
dato^=0x80;
```

È interessante notare che se questa istruzione viene richiamata una seconda volta, porterà il bit al suo valore originale.

Operatore NOT (~)

Questo operatore, a differenza degli altri, è un operatore "unario", cioè prevede un solo operando. Quando il simbolo ~ (si legga *tilde*) è anteposto ad un'espressione, tutti i bit vengono invertiti (negati). Se ad esempio la variabile `b` (`unsigned char`) vale 0x03 (in binario 00000011), allora il valore di `~b` sarà 0xFC (in binario 11111100).

Operatori di scorrimento (<<, >>)

Questi operatori consentono di scorrere i bit di una variabile a sinistra (<<) o a destra (>>) di un certo numero di posizioni. Ad esempio per il programma:

```
main() {
  unsigned char a;

  a = 15;

  printf("a << 1 = %d\n", a<<1);
  printf("a << 2 = %d\n", a<<2);
  printf("a >> 1 = %d\n", a>>1);
}
```

l'output sarà il seguente:

```
a << 1 = 30
a << 2 = 60
a >> 1 = 7
```

Considerando la rappresentazione binaria dei 4 numeri coinvolti, l'effetto sui dati dovrebbe risultare molto intuitivo:

```
00001111  (15)
00011110  (30, cioè 15<<1)
```

```
00111100  (60, cioè 15<<2)
00000111  (7,  cioè 15>>1)
```

Si nota che i bit che “escono” dal lato dell’MSB o dell’LSB vengono semplicemente perduti, mentre i bit che “entrano” da entrambi i lati sono degli 0.
Nel caso di variabili con segno, il cui MSB vale 1, si potrebbe avere un’introduzione di 1 dal lato dell’MSB in casi di scorrimento a destra. Questo corrisponde ad uno scorrimento “aritmetico”, che conserva cioè il segno nella rappresentazione in complemento a due (caratteristica utile quando vengono manipolati valori numerici). Dall’esempio si nota anche che gli scorrimenti a destra ed a sinistra corrispondono rispettivamente a divisioni o moltiplicazioni per potenze di due (2, 4, etc.).

7. Array, stringhe e strutture

GLI ARRAY

Gli array sono lo strumento per la gestione delle variabili indicizzate, variabili che possono contenere diversi valori ciascuno dei quali è identificato da un indice. Ad esempio per memorizzare la quantità di pioggia caduta in ogni mese è possibile utilizzare un array, piuttosto che 12 variabili separate. L'indice dell'array in questo caso rappresenterebbe proprio il mese, mentre il contenuto dell'array i mm di pioggia caduti in quel particolare mese. Il programma seguente mostra come utilizzare un array per richiedere all'utente e memorizzare proprio queste informazioni:

```
main() {
  int pioggia[12];
  int i;

  for(i=0; i<12; i++) {
    print("Pioggia nel mese %d: ", i+1);
    scanf("%d", &pioggia[i]);
  }

  …

}
```

Nell'esempio l'array `pioggia` è stato dichiarato similmente ad normale variabile di tipo `int`, ma è stato indicato tra parentesi quadre il numero di elementi da cui sarà composto (in questo caso 12). È possibile creare array di variabili di qualsiasi tipo, anche definito dall'utente (si veda di seguito). L'indice di un array può variare da zero fino al numero di elementi diminuito di uno (quindi da 0 a 11 nel caso dell'esempio). Il compilatore non esegue nessun controllo sugli indici utilizzati nel il programma quindi utilizzando un indice maggiore del numero di elementi dell'array, non verranno segnalati errori, ma il programma potrebbe non funzionare correttamente!
È possibile dichiarare un array ed inizializzarne contemporaneamente il contenuto:

```
main() {
  int dati[4]={5, 403, 123, -43};
```

```
  int i;

  for(i=0; i<4; i++) {
    printf("Dato in posizione %d: %d\n", i, dati[i]);
  }
}
```

Esiste anche la possibilità di potere dichiarare array a più dimensioni, cioè con più indici:

```
main() {
  int dati[4][2]={{5,1}, {403,3}, {123,9}, {-43,2}};
  int i;

  for(i=0; i<4; i++) {
    printf("%d - %d\n", dati[i][0], dati[i][1]);
  }
}
```

La dimensione di un array deve essere stabilita in fase di scrittura del programma e, a differenza di quanto avviene in altri linguaggi, non può essere cambiata a *run-time* (cioè mentre il programma è in esecuzione). Utilizzando gli array è necessario dunque accertarsi che la dimensione di eventuali dati acquisiti a run-time sia compatibile con quella dell'array dichiarato. In realtà, come verrà spiegato meglio nel capitolo 9, gli array offrono delle possibilità molto più ampie ed avanzate di quelle descritte in questo capitolo ed è quindi possibile aggirare facilmente questa limitazione.

LE STRINGHE

In molte applicazioni emerge la necessità di memorizzare o di elaborare del testo, delle parole o delle frasi. La maggior parte dei linguaggi ad alto livello mette a disposizione strumenti per gestire queste stringhe di testo (di seguito semplicemente "stringhe"). Il linguaggio C non fa eccezione, pur mantenendo lo stile essenziale rivolto al basso livello. In C le stringhe sono rappresentate da array di `char`. Ciascun elemento dell'array rappresenta (o memorizza) un carattere della stringa, codificato secondo il codice ASCII, come già spiegato nel capitolo 2. Ad esempio la stringa "Ciao" potrà essere memorizzata in un array di `char` (o `unsigned char`) il cui primo elemento sarà 'C' (codice ASCII 67), il secondo sarà 'i' (codice ASCII 105) e così via (la tabella dei codici ASCII è riportata nell'appendice A). In realtà tutte le stringhe devono essere "terminate", aggiungendo dopo l'ultimo carattere il valore decimale 0 (a cui corrisponde un codice ASCII non stampabile), detto appunto "terminatore di stringa". Per stampare le stringhe memorizzate con la funzione `printf`, occorre usare il carattere di conversione %s:

```
main() {
  char stringa[6] = {'C', 'i', 'a', 'o', '!', 0};

  printf("%s", stringa);
}
```

Si noti che l'array è stato dichiarato di lunghezza 6, perché è necessario includere il terminatore, inoltre quando la variabile associata alla stringa deve essere utilizzata, non occorre specificare l'indice. Anche se è stato inserito il testo usando dei caratteri ASCII tra apici, il contenuto dell'array è sempre rappresentato dai corrispondenti numeri del codice, come mostra il seguente esempio:

```
main() {
  char stringa[6] = "Ciao!";

  stringa[0]=77; // Codice ASCII del carattere 'M'

  printf("%s", stringa);
}
```

È possibile anche visualizzare in forma numerica il contenuto di una stringa:

```
main() {
  char stringa[6] = "Ciao!";
  int i;

  for(i=0; i<sizeof(stringa); i++)
    printf("stringa[%d]=%d\n", i, stringa[i]);
}
```

L'esempio precedente genera il seguente risultato (l'elenco dei codici ASCII relativi al testo memorizzato):

```
stringa[0]=67
stringa[1]=105
stringa[2]=97
stringa[3]=111
stringa[4]=33
stringa[5]=0
```

Le stringhe possono anche essere lette dallo standard input (quindi acquisite e memorizzate a run-time):

```
main() {
  char str[32];

  printf("Stringa: ");
  scanf("%s", str);

  printf("\nInserito: %s", str);
}
```

Nell'esempio precedente l'array è stato dichiarato largo 32 elementi, quindi il testo più lungo che può essere inserito è di 31 caratteri (uno è per il terminatore). Poiché la funzione `scanf` non esegue nessun tipo di controllo sul testo immesso, scrivendo l'array oltre il 32-esimo

elemento, viene coinvolta una zona di memoria non appropriata. Questo fenomeno è noto come *buffer overrun* e può provocare malfunzionamenti, il blocco del programma o addirittura del sistema. Inoltre è importante ricordare che la funzione `scanf` interrompe la memorizzazione quando incontra uno spazio nella stringa (si può provare facilmente usando l'ultimo esempio).
La libreria *strings.h* contiene specifiche funzioni per il trattamento delle stringhe. Tali funzioni verranno analizzate nel capitolo 8.

LE STRUTTURE

Oltre ai tipi di dati nativi, il linguaggio C offre la possibilità di creare dei tipi di dati definiti dall'utente. Per fare questo è possibile utilizzare l'istruzione `typedef`. Ad esempio per definire il tipo `Byte`, che sarà semplicemente un `unsigned char` e il tipo `Nome` come un array di 32 caratteri:

```
#include <stdio.h>

typedef unisgned char Byte;
typedef char[32] Nome;

main() {
  Nome stringa;
  Byte contatore;

  scanf("%s", stringa);
  …
}
```

I tipi definiti possono essere utilizzati, come quelli standard, per dichiarare delle variabili o eseguire dei casting. Un secondo metodo per definire dei nuovi tipi in C è quello di creare variabili strutturate. Per dichiarare una variabile strutturata viene usato il costrutto `struct`, come mostrato nell'esempio seguente:

```
struct oggetto {
  int volume;
  int peso;
  char nome[32];
};

main() {
  struct oggetto obj;

  obj.volume=2;
  obj.peso=15;
  obj.nome="dado";
  …
}
```

Nel costrutto `struct` viene dichiarato il nome della struttura (in questo caso `oggetto`) quindi vengono elencati i campi, ciascuno dei quali può essere di qualsiasi tipo. Per utilizzare il nuovo tipo strutturato occorre prima istanziare una variabile di quel tipo (in questo caso `obj`) quindi accedere ai suoi campi usando la sintassi *variabile.campo* (il punto separa la variabile dal campo selezionato). Sarebbe anche stato possibile definire un array di variabili di tipo `oggetto`:

```
…
   struct oggetto obj[10];

   obj[0].volume=2;
   obj[0].peso=15;
   obj[0].nome="dado";
…
```

Spesso il costrutto `struct` viene associato al `typedef`, nel seguente modo:

```
typedef struct oggetto {
   int volume;
   int peso;
   char nome[32];
};
```

In questo modo le variabili potranno essere dichiarate senza usare la keyword `struct`:

```
oggetto obj[10];
```

Una possibilità interessante, da usare con attenzione soprattutto se non si conosce a fondo il comportamento del compilatore, è quella di dichiarare variabili strutturate i cui campi sono formati da un numero di bit arbitrario:

```
typedef struct pacchetto
{
   char sottotipo :4;
   char tipo :2;
   char criptato :1;
   unsigned short lunghezza;
   unsigned char dati[2346];
};
```

Questo tipo strutturato potrebbe descrivere il pacchetto utilizzato in un ipotetico protocollo di comunicazione, in cui i primi 4 bit rappresentano il sottotipo di pacchetto, i successivi 2 il tipo, il successivo se il pacchetto è criptato o meno. Seguono due campi "normali" in cui è riportata la lunghezza dei dati ed i dati veri e propri, contenuti in un array.

I TIPI ENUMERATIVI

Il tipo "enumerativo" (`enum`), permette di associare ad un insieme di valori interi (`int`) degli identificativi mnemonici. Ad esempio se una variabile viene utilizzata per memorizzare il gior-

no della settimana è molto più intuitivo indicarne i valori con i simboli “Lun”, “Mar”, “Mer”, …“Dom”, piuttosto che con i numeri 0, 1, 2, …6. Il compilatore assocerà agli identificativi i loro valori numerici in maniera trasparente all’utente. Per dichiarare un tipo enumerativo si procede come nel seguente esempio:

```
#include <stdio.h>

main() {
  enum Giorni {Lun, Mar, Mer, Gio, Ven, Sab, Dom} giorno;

  giorno=Sab;

  giorno++;

  if (giorno==Dom)
    printf("Oggi e’ Domenica!\n");

  printf("giorno = %d\n", giorno);

  getchar();
}
```

Nell’esempio è stato creato il tipo enumerativo `Giorni` ed è stata istanziata la variabile `giorno`. Alla variabile è stato assegnato il valore `Sab`, che è stato poi incrementato, assumendo così il valore `Dom` (come dimostra il successivo confronto). Visualizzando il contenuto della variabile con una `printf` verrà ottenuto il valore 6 e non l’identificativo. Questo accade perché, come già detto, le variabili enumerative sono trattate dal compilatore come degli `int` ed ai valori mnemonici è associato un valore numerico crescente.
È possibile specificare esplicitamente il valore da associare a ciascun simbolo:

```
enum Opzioni {Ok=3, Annulla=0, Cancella, Riprova=-1} opzione;
```

Quando non viene specificato un valore, il compilatore assegna il valore del precedente elemento incrementato di uno. L’uso dei tipi enumerativi permette di ottenere un codice più leggibile e più semplice da aggiornare o modificare.

8. Le funzioni di libreria

INTRODUZIONE

Come già visto nei precedenti capitoli, molte funzioni utili dell'ANSI C non sono parte integrante del linguaggio, ma sono disponibili sotto forma di librerie che vanno richiamate nel proprio programma utilizzando la direttiva `#include`. Il motivo di questa scelta risiede nel fatto che queste funzioni sono in genere strettamente legate all'hardware e/o dipendenti della macchina su cui funzionerà il programma. Il fatto di separare queste funzioni dal linguaggio vero e proprio permette di rendere i programmi in gran parte indipendenti dall'hardware, favorendo quindi la portabilità del codice. Utilizzando lo stesso meccanismo è inoltre possibile aggiungere anche funzioni personalizzate o sostituire quelle esistenti per ottimizzare maggiormente le prestazioni qualora fosse necessario. In questo capitolo verranno analizzate le principali funzioni standard e le librerie che le compongono.

FUNZIONI DI INPUT E OUTPUT

L'ANSI C non dispone di istruzioni per comunicare con il mondo esterno, perfino la "fondamentale" funzione `printf` deve essere richiamata da una libreria. La libreria in questione, richiamata con un `#include` in tutti gli esempi visti in precedenza, è la *stdio.h* ("standard I/O") e, come suggerisce il nome, contiene funzioni utili per comunicare e scambiare dati con il mondo esterno. In realtà i file con estenzione *.h* richiamati dal programma non sono le librerie vere e proprie, ma dei file chiamati *header* che contengono la dichiarazione delle funzioni messe a disposizione dalla libreria e le relative costanti. Le librerie vere e proprie sono disponibili di solito sotto forma di codice già compilato, pronto per essere "inglobato" nei programmi (tramite un'operazione di *linking*). Le funzioni messe a disposizione dalla libreria *stdio.h* permettono di comunicare con i dispositivi dello *standard input* e *standard output* (dispositivi a caratteri: di solito lo schermo e la tastiera sui PC o una porta seriale sulla maggior parte dei sistemi embedded) e di gestire i file. La funzione in assoluto più utilizzata della libreria è la `printf` già analizzata in precedenza.
La funzione complementare di `printf` è la `scanf`. Questa funzione permette di leggere i dati dallo standard input e di assegnarli alle variabili. La sintassi è la seguente:

```
int scanf("stringa di formattazione", elenco variabili…);
```

La funzione permette anche di “de-formattare” il testo letto e di ripartirlo nella lista di variabili indicata. Ecco un esempio di utilizzo:

```
#include <stdio.h>

main() {
  /* dichiarazione variabili */
  char testo[32];
  int num;

  /* richiede dati */
  scanf(“%31s %d”, stringa, &num);

  /* visualizza dati */
  printf(“\nStringa = %s”, testo);
  printf(“\nIntero = %d”, num);
}
```

La `scanf` non permette di visualizzare un testo con la richiesta di immissione di dati, attende solamente che i dati indicati nella stringa di formattazione vengano immessi (si può ovviare scrivendo la richiesta con una `printf`). La `scanf` inoltre non prende come argomento le variabili, ma il loro indirizzo! Occorre cioè passare le variabili per riferimento. Questo, come già visto, implica l’utilizzo del carattere “&” nel caso variabili normali, oppure indicando semplicemente il nome nel caso di puntatori o array (dal momento che questi elementi rappresentano già un indirizzo).
Così come la funzione `printf`, la `scanf` ha un’infinità di caratteristiche e funzionalità aggiuntive, per le quali si rimanda all’appendice B ed alle guide degli specifici compilatori.
Altre due funzioni simili a `printf` e `scanf`, ma più semplici da utilizzare, sono `getc` e `putc`. La sintassi di queste funzioni è la seguente:

```
int getc(FILE *stream);
int putc(int c, FILE *stream);
```

Queste funzioni permettono di inviare o leggere singoli caratteri (nonostante il tipo utilizzato sia un intero), sia da file che dai dispositivi dello standard I/O. Questo le rende particolarmente adatte nel caso in cui lo standard I/O è rappresentato ad esempio da una porta di comunicazione seriale. Un esempio di utilizzo di queste funzioni è il seguente:

```
#include <stdio.h>

main() {
  int c;

  /* ciclo infinito */
  while(1) {
    /* legge carattere */
    c = getc(stdin);
    /* scrive carattere */
    if (c!=EOF)
```

```
        putc(c, stdout);
    }
}
```

Il programma legge un carattere dallo standard input e lo scrive sullo standard output. Nel caso in cui questi dispositivi siano rappresentati dallo schermo e dalla tastiera, verranno visualizzati i caratteri digitati, mentre se lo standard input ed output è rappresentato da una periferica seriale il programma invierà il carattere ricevuto.
Da notare che le funzioni richiedono di specificare uno "stream" per i dati. Nell'esempio sono stati specificati quelli di standard I/O, utilizzando due costanti contenute nella libreria *stdio.h*: `stdin` ed `stdout`. Altre costanti importanti definite nella libreria sono `NULL` ed `EOF`, che specificano particolari valori associati a puntatori vuoti o al carattere di fine file ("End Of File"). La funzione `getc` restituisce proprio quest'ultimo carattere se lo stream è vuoto o se si è verificato un errore (da qui l'`if` che compare nel codice dell'esempio precedente).

Gestione dei file

Per il C un file è uno stream di dati, così come gli standard I/O. Per utilizzare i file è necessario dichiarare un apposito puntatore al tipo `FILE` (definito nella *stdio.h*).
È necessario quindi aprire il file con la funzione `fopen`, eseguire le operazioni di lettura o scrittura quindi chiudere il file con `fclose`. Le funzioni utilizzabili per la lettura e la scrittura sono molte, tra queste vale la pena citare le già viste `getc` e `putc` (e le varianti `fgetc` e `fputc`), usate per leggere e scrivere singoli caratteri (cioè singoli byte) e le funzioni `fprintf` ed `fscanf`, che funzionano esattamente come `printf` e `scanf`, ma operano su file di testo. La sintassi di queste funzioni è la seguente:

```
FILE *fopen("nome file", "stringa modo");
int fprintf(FILE *stream, "stringa formato" , elenco variabili…);
int fscanf(FILE *stream, "stringa formato", elenco variabili…);
int fclose(FILE *stream);
```

La funzione `fopen` ha due stringhe come argomenti, la prima indica il nome del file da aprire (eventualmente comprensivo di percorso), la seconda specifica la modalità di apertura del file. Un file può essere aperto in modalità lettura (stringa "r"), scrittura (stringa "w"), binario in lettura o scrittura ("rb" e "wb") o in modalità "append" ("a") in cui i dati vengono scritti di seguito a quelli esistenti. In caso di successo la funzione restituisce il puntatore al file aperto o in caso di errore o insuccesso il valore `NULL`.
Le funzioni `fprintf` ed `fscanf` funzionano come le loro analoghe, ma richiedono anche il puntatore al file aperto coinvolto nella lettura o scrittura. Entrambe scrivono o leggono dal file una riga di testo alla volta. Una particolarità di tutte queste funzioni è che la lettura e la scrittura in un file avvengono in modo sequenziale, cioè ogni volta viene utilizzata una funzione, il puntatore al file viene incrementato e l'operazione successiva viene quindi eseguita sul carattere o sulla riga successiva. Il puntatore viene riportato all'inizio del file con la funzione `rewind`:

```
void rewind(FILE *stream);
```

Per posizionare il puntatore in una posizione qualsiasi è disponibile invece la funzione `fseek`:

```
int fseek(FILE *stream, long offset, int whence);
```

La variabile **offset** indica la posizione (in numero di byte) in cui spostare il puntatore, mentre la variabile **whence** specifica da quale punto deve essere inteso il conteggio (0 = inizio del file, 1 = posizione corrente, 2 = file del file).
Di seguito è riportato esempio che illustra l'uso delle funzioni viste e che può risultare anche utile in pratica: il programma legge un file di testo in formato UNIX/Linux (carattere ASCII di fine linea 0x0A), lo visualizza a schermo e lo salva in formato Dos/Windows (caratteri di fine linea 0x0D 0x0A).

```
#include <stdio.h>

main() {
  char *nomefine_in = "Percorso/nome1.txt";
  char *nomefine_out = "Percorso/nome2.txt";
  int c, i;
  FILE *finein, *fileout;

  /* apertura file */
  filein=fopen(nomefile_in, "r");
  fileout=fopen(nomefile_out, "w");

  while((c=getc(filein))!=EOF) {

    /* stampa a schermo */
    printf("%c", c);

    /* controlla tipo di carattere */
    if (c!=0x0A) {
      /* testo normale */
      putc(c, fileout);
    } else {
      /* fine linea */
      putc(0x0D, fileout);
      putc(0x0A, fileout);
    }
  }

  /* chiusura file */
  fclose(filein);
  fclose(fout);

}
```

Il programma apre i due file in modalità lettura e scrittura rispettivamente (quello di scrittura viene creato automaticamente se non esiste già) ed esegue un loop per leggere tutti i caratteri, fino a quando non viene restituito il carattere di fine file **EOF** (la lettura e l'assegnazione viene fatta direttamente nella condizione del **while**). Il carattere letto viene controllato per capire se si tratta di un carattere normale o di un "a capo": nel primo caso

viene copiato sul file di destinazione, altrimenti viene rimpiazzato con i due previsti dal formato Dos/Windows. Terminato il loop vengono chiusi i file prima di uscire dal programma. Per rendere il programma più completo ed affidabile sarebbe opportuno controllare che le due funzioni `fopen` restituisca un valore valido, in caso contrario (errore di apertura) uscire senza compiere alcuna operazione o comunque segnalando all'utente il problema (usando una `printf`).

Per leggere o scrivere un file in modalità binaria, analogamente vengono coinvolti blocchi di dati e non singoli byte. Per fare questo sono disponibili le funzioni `fread` ed `fwrite`:

```
long fread(void *ptr, long size, long n, FILE *stream);
long fwrite(void *ptr, long size, long n, FILE *stream);
```

Queste prendono in ingresso il puntatore ad un blocco di memoria in cui si trovano o andranno scritti i dati (ad esempio un array), la lunghezza in byte del blocco da leggere o scrivere, il numero di blocchi (consecutivi) su cui operare ed il puntatore al file. La distinzione tra dimensione del blocco e numero di blocchi può tornare utile per operare con delle variabili strutturate memorizzate consecutivamente. Un esempio di utilizzo delle due funzioni è il seguente:

```
long dati_out[16];
long dati_in[16];
FILE *f_output, *f_input;
…
fwrite(dati_out, sizeof(long), 16, f_output);
…
fread(dati_in, 16*sizeof(long), 1, f_input);
```

Va notato che a scopo di esempio sono state utilizzate due scritture diverse, ma le operazioni sono equivalenti: in un caso è stata specificata la lunghezza di un elemento ed è stata richiesta la scrittura di 16 elementi, nell'altro è stata direttamente calcolata la lunghezza complessiva dei dati ed è stata richiesta la lettura di un solo blocco che li comprende tutti. Il risultato è identico: i dati (in questo caso 4*16=64 byte) sono sempre letti e scritti in sequenza, sia nel file che nel buffer di memoria.

FUNZIONI MATEMATICHE

Capita spesso di dovere utilizzare nei propri programmi funzioni matematiche più complesse rispetto alle normali operazioni aritmetiche native del linguaggio. L'ANSI C mette a disposizione una libreria in cui sono state definite tutte le più comuni funzioni matematiche avanzate: la libreria *math.h*.
Le funzioni contenute nella libreria sono riportate nella tabella 8.1, sono disponibili le funzioni trigonometriche dirette, inverse ed iperboliche, quelle relative ai logaritmi, esponenziali, potenze, radici quadrate e valori assoluti. La maggior parte di queste funzioni utilizza variabili di tipo floating point (`float`, `double`, `long double`), questo implica che se vengono utilizzati tipi interi come argomenti o come risultati viene fatta una conversione (*casting*) automatica in ingresso o in uscita, che può comportare una perdita di precisione. Inoltre è

probabile che su sistemi più piccoli (a microcontrollore), non tutte le funzioni siano supportare, oppure sono supportare solo con tipi interi e con un fissato numero di bit. Per questo motivo prima di utilizzarle è bene leggere la documentazione del compilatore.

FUNZIONE	SINTASSI	DESCRIZIONE
abs	`int abs(int x);`	Valore assoluto di X
acos	`double acos(double x);`	Arco-coseno di X
asin	`double asin(double x);`	Arco-seno di X
atan	`double atan(double x);`	Arcotangente di X
atan2	`double atan2(double y, double x);`	Arcotengente di y/x
atof	`double atof(char *s);`	Converte la stringa s in float
ceil	`double ceil(double x);`	Arrotonda all'intero (> X)
cos	`double cos(double x);`	Coseno di X
cosh	`double cosh(double x);`	Coseno iperbolico di X
exp	`double exp(double x);`	Esponenziale eX
fabs	`double fabs(double x);`	Valore assoluto di X (float)
floor	`double floor(double x);`	Arrotonda all'intero (< X)
fmod	`double fmod(double x, double y);`	X modulo Y (float)
log	`double log(double x);`	Logaritmo naturale di X
log10	`double log10(double x);`	Logaritmo decimale di X
pow	`double pow(double x, double y);`	X elevato a Y
sin	`double sin(double x);`	Seno di X
sinh	`double sinh(double x);`	Seno iperbolico di X
sqrt	`double sqrt(double x);`	Radice quadrata di X
tan	`double tan(double x);`	Tangente di X
tanh	`double tan(double x);`	Tangente iperbolica di X

Tabella 8.1 *Le funzioni matematiche della libreria math.h*

LIBRERIA STDLIB.H

Una libreria che contiene molte funzioni importanti (in certi casi fondamentali) è la *stdlib.h*. Una prima classe di funzioni contenute in questa libreria riguarda le operazioni su numeri interi, un'altra classe riguarda la generazione dei numeri casuali e un'altra ancora l'allocazione dinamica della memoria e le funzioni legate al sistema.
Al primo gruppo, appartengono le utilissime funzioni per convertire le stringhe in numeri: `atoi` ed `atol` (che convertono una stringa in un intero rispettivamente di tipo `int` o `long`). L'equivalente per numeri floating point è `atof` della libreria *math.h*. Il formato di queste funzioni è il seguente:

```
int atoi(char *s);
long atol(char *s);
```

Le funzioni che riguardano la generazione di numeri casuali sono `rand` ed `srand`. La prima restituisce un numero intero estratto (con distribuzione di probabilità uniforme) da una sequenza pseudo-casuale, nel range da 0 a `RAND_MAX` (costante definita nella libreria). La seconda funzione invece consente di inizializzare il generatore di numeri casuali con un certo "seme" (questa possibilità è utile per creare una sequenza pseudo-casuale ripetibile). Per ricondurre il numero casuale ottenuto in un particolare range di valori è possibile utilizzare l'operatore modulo "%" (che restituisce il resto di una divisione intera) o gli operatori binari. Ecco un esempio:

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

main() {
  int i;

  /* inizializzazione sequenza */
  srand(5);

  printf("\nSequanza di lanci di un dado:");
  for(i=0; i<10; i++)
   printf("\n%d", (rand()  % 6 + 1));
  }
}
```

Il programma calcola 10 numeri casuali compresi tra 1 e 6, imitando il lancio di un dado. Dal momento che il generatore viene inizializzato sempre con lo stesso seme all'avvio, la sequenza sarà sempre uguale. Per ottenere un numero da 1 a 6 è stato sufficiente usare l'operatore modulo, ottenendo così un numero compreso tra 0 e 5, a cui è stato sommato 1.

Altre importantissime funzioni presenti nella libreria sono quelle relative all'allocazione dinamica della memoria. Tra queste meritano un cenno la `malloc` e la `free`. Grazie a queste funzioni è possibile richiedere una certa quantità di memoria (espressa in byte) per memorizzare dei dati per poi rilasciarla quando non serve più. I possibili usi di queste funzioni sono molteplici, ma richiedono conoscenze approfondite per essere compresi al meglio. Tuttavia vale la pena mostrare un esempio molto interessante del loro utilizzo. In un'applicazione in cui non è possibile conoscere a priori l'esatta dimensione dei dati è possibile aggirare il problema sovradimensionando l'array. Esiste però un modo molto più elegante ed efficiente: allocare l'array solo quando se ne conosce l'esatta dimensione!
Questa tecnica è mostrata nell'esempio seguente:

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

main() {
  int i, dim;
  int *blocco_mem;
```

```
    /* richiesta all'untente del numero
       di elementi da inserire */
    printf("Numero elementi?\n");
    scanf("%d", &dim);

    /* allocazione dell'array */
    blocco_mem = malloc(dim * sizeof(int));

    /* allocazione non riuscita? */
    if (blocco_mem == NULL) {
      printf("\nErrore di allocazione!!!");
      exit(EXIT_FAILURE);
    }

    /* immissione dei dati */
    for(i=0; i<dim; i++) {
      printf("Inserisci numero?\n");
      scanf("%d", blocco_mem + i);
    }

    /* visualizzazione dei dati */
    printf("Dati immessi: numero?\n");
    for(i=0; i<dim; i++)
      printf("%d\n", blocco_mem[i]);

    /* rilascia la memoria allocata */
    free(blocco_mem);
}
```

Il programma chiede all'utente il numero dei dati da inserire, alloca un array delle giuste dimensioni usando la funzione `malloc`. Se l'allocazione non dovesse riuscire (per mancanza di memoria o altro) il programma termina. Per fare questo viene utilizzata un'altra funzione della *stdlib.h:* `exit`. La costante `EXIT_FAILURE` è definita nella libreria e, con la `EXIT_SUCCESS`, serve per indicare all'eventuale sistema operativo o ad un processo padre, se il programma è terminato in maniera normale o a causa di un errore. Da notare che la variabile per il blocco di memoria è stata dichiarata come un puntatore ad interi, a cui è stato poi assegnato l'indirizzo del blocco allocato restituito dalla `malloc`. Questo dimostra che i puntatori possono essere poi utilizzati come dei normali array, aspetto che verrà approfondito nel prossimo capitolo.

9. I puntatori

I PUNTATORI

I puntatori rappresentano sicuramente uno degli aspetti meno intuitivi, ma al tempo stesso più utili e potenti del linguaggio C. Come già detto i puntatori sono variabili che contengono gli indirizzi di memoria di altre variabili.Per i programmatori assembler questo non risulterà una grossa novità, è abbastanza frequente infatti in questi casi avere a che fare con modi di indirizzamento indiretti o indicizzati, oppure utilizzare esplicitamente degli indirizzi di memoria noti per memorizzare le variabili. Il concetto è lo stesso, ma in questo caso buona parte dei meccanismi a più basso livello sono gestiti dal compilatore (quindi sono trasparenti al programmatore). Il potere conoscere la locazione di memoria di una variabile in C risulta fondamentale quando una funzione deve modificare una variabile anziché restituire semplicemente un valore e consente ad una funzione di restituire più di una variabile alla volta, come nell'esempio seguente:

```
#include <stdio.h>

/* Funzione divisione */
void Divisione(int dd, int ds, int *q, int *r)
{
  /* Calcolo quozionete */
  *q = dd / ds;

  /* Calcolo resto */
  *r = dd % ds;
}

main()
{
  int dvd, div, quoz, res;

  dvd = 10;
  div = 4;
  Divisione(dvd, div, &quoz, &res);

  printf("%d / %d = %d \n Resto = %d", dvd, div, quoz, res);
}
```

La funzione `Divisione` prende in ingresso due numeri interi (il divisore ed il dividendo) e due puntatori alle variabili in cui verranno memorizzati i risultati (il quoziente ed il resto). Per passare l'indirizzo delle variabili viene utilizzato l'operatore & posto prima del nome della variabile stessa. Per indicare il contenuto della locazione di memoria indirizzata da un puntatore viene utilizzato l'operatore *.
Le variabili `q` ed `r` all'interno della funzione `Divisione` conterranno gli indirizzi di memoria delle variabili `quoz` e `res` utilizzate nel main ed andranno utilizzate come puntatori. Lo stesso meccanismo può essere utilizzato anche nel caso di variabili di tipo strutturato; il funzionamento è identico, ma è possibile utilizzare in questo caso una notazione semplificata per indicare il contenuto di un campo particolare, come mostrato nell'esempio seguente:

```
…
typedef struct {
  int ora;
  int minuti;
  int secondi;
} tempo;

…
void Resetta(tempo *t);
{
/* la notazione estesa sarebbe:
(*t).ora = 0;
(*t).minuti = 0;
(*t).secondi = 0;
*/

/* notazione compatta: */
t->ora = 0;
t->minuti = 0;
t->secondi = 0;
}
```

L'uso dei puntatori può indurre facilmente in errore. Si consideri ad esempio il codice seguente:

```
main()
{
  /* dichiarazione intero */
  int a;
  /* dichiarazione puntatori */
  int  *f, *g;

  a=5;
  f=&a;

  g=f;
  *g=2;
```

```
  printf("a vale: %d", a);
}
```

Quale sarà il valore finale di `a`? Contrariamente a quello che si potrebbe pensare ad una lettura superficiale, alla fine `a` vale 2! Questo perché ponendo `g = f`, `g` va a puntare la stessa locazione di `f`, che è quella in cui si trova il valore di `a` . Modificando il contenuto della locazione puntata da `g`, in realtà si modifica proprio `a`! Altri errori comuni consistono nel passare una variabile ad una funzione anziché il suo indirizzo (quindi dimenticare di anteporre la &), oppure utilizzare all'interno della funzione un puntatore come se fosse una variabile. La probabilità di commettere questi errori aumenta notevolmente nell'uso di puntatori doppi o comunque multipli (un puntatore è doppio quando punta ad una variabile che è a sua volta un puntatore).

Se venisse inizializzato il valore di `g` o `f` anziché `a` si otterrebbe un numero, che è proprio l'indirizzo di memoria in cui si trova la variabile `a`. Le caratteristiche di questo numero dipendono da quelle del sistema per cui è stato compilato il programma e dal modello di memoria utilizzato: cambiandolo manualmente anziché leggere il valore di `a`, verrebbe letto il valore di un'altra locazione di memoria. Questo particolare, apparentemente privo di importanza si rivela invece fondamentale nel caso di programmazione di piccoli sistemi, dove la gestione della memoria è pressoché manuale: il linguaggio C permette in questo modo di leggere e scrivere in locazioni di memoria a piacere, semplicemente attribuendo manualmente il valore ad un puntatore. L'esempio che segue legge il contenuto di 20 locazioni di memoria consecutive, a partire da un indirizzo specificato (in questo caso 0x0000).

```
#include <stdio.h>

main()
{
     int i, BASE_ADDR=0x0000;
     char *punt;

     for(i = 0; i<20; i++)
{
           punt = BASE_ADDR + i;
printf("%X: %X\n", punt, *punt);
     }
}
```

Questo test non è molto significativo se eseguito su un recente PC, in quanto la complessa gestione della memoria (sia delle macchine x86, che del sistema operativo), nascondono o falsano un po' i risultati.
Su sistemi più piccoli il test può però risultare molto più utile e significativo, infatti in modo simile è possibile per esempio assegnare ad un puntatore l'indirizzo di una porta di I/O e scrivere o leggere la porta semplicemente scrivendo e leggendo la variabile associata, come nell'esempio che segue:

```
main()
{
  /* dichiarazione puntatore */
  char *porta;
```

```
   /* assegnazione indirizzo al puntatore */
   porta = (char *) 0xFF00;

   /* scrittura valore 4 nella porta */
   *porta = 4;

   /* lettura valore porta */
   printf("Valore porta: %d", *porta);
}
```

In questo caso è stato supposto che il sistema abbia uno spazio di indirizzamento della memoria a 16 bit e che le periferiche siano mappate su questo spazio.
Un modo più semplice ed efficiente di utilizzare questa possibilità consiste nel dichiarare come macro i puntatori agli indirizzi che interessano:

```
#define PORTA1  *((char *) 0xFF00)
#define PORTA2  *((char *) 0xFF01)

main() {
   // scrittura nella Porta1
   PORTA1=0xFA;

   // Lettura dalla Porta2
   printf("Contenuto Porta1: &X", PORTA2);
}
```

In questo modo sarà possibile utilizzare le macro come normali variabili per eseguire letture e scritture. Il codice associato alle due macro infatti, nel linguaggio dei puntatori, significa "contenuto della locazione 0xFF0x, dove 0xFF0x è un puntatore a `char`". Spesso i compilatori dedicati ad alcuni particolari sistemi forniscono un file header (.h) da includere, in cui sono già dichiarate tutte le macro per i più importanti registri e locazioni di memoria.

È lecito chiedersi per quale motivo è necessario specificare un tipo di dati per i puntatori (`char` nell'esempio) dal momento che essi rappresentano degli indirizzi e quindi hanno un formato predefinito dipendente solo dalle caratteristiche della macchina. Il motivo risiede nel fatto che l'informazione sul tipo di dati viene utilizzata quando vengono compiute delle operazioni aritmetiche sui puntatori: ad esempio se i dati sono di tipo `char`, un incremento di 1 indica che anche l'indirizzo deve essere incrementato di 1 byte, se i dati sono di tipo `long`, l'incremento sarà di 4 byte. Va ricordato comunque che possono essere dichiarati anche puntatori al tipo `void`, che vengono in genere utilizzati per puntare a tipi non definiti a priori o quando il puntatore è usato per indicare un indirizzo ben preciso (e per questo indipendente dal tipo di dati).

GLI ARRAY

Nel linguaggio C è possibile creare matrici di elementi utilizzando gli array. L'utilizzo degli array, come visto nei precedenti capitoli è molto semplice ed intuitivo e non presenta particolari difficoltà. Quello che non sempre è noto e che invece può essere molto utile, è il meccanismo con cui sono implementati in C gli array. Quando viene dichiarato un array il compilatore alloca un'area di memoria di dimensione tale da contenere tutti gli elementi richiesti. Gli elementi dell'array saranno memorizzati in quest'aera in successione ordinata. La variabile associata all'array è un normale puntatore che indica la locazione di memoria corrispondente al primo elemento dell'array stesso. Le parentesi quadre *[ ]*, utilizzate per racchiudere l'indice, sono un operatore che ha la funzione di sommare all'indirizzo del primo elemento un offset proporzionale al valore dell'indice. In definitiva la scrittura `nomearray[i]` equivale all'espressione `*(nomearray+i)`. Dovrebbe risultare abbastanza evidente che è possibile utilizzare gli array ed i puntatori in maniera intercambiabile, a seconda di cosa sia più conveniente. Ad esempio per passare un array ad una funzione è sufficiente passare un puntatore, utilizzando nella chiamata il solo nome dell'array. All'interno della funzione l'array potrà essere utilizzato nel modo consueto, come visibile nell'esempio seguente:

```
…
void Azzera_Dati(float *d, int n)
{
  int i;

  for(i=0; i<n; i++)
    d[i]=0;
}

main()
{
  float dati[32];

  Azzera_Dati(dati, 32);
…
}
```

La funzione `Azzera_Dati` accetta un array (sotto forma di puntatore) ed un intero che indica da quanti elementi è composto l'array quindi ne azzera tutti gli elementi con un semplice ciclo `for`. La funzione è richiamata indicando semplicemente il nome dell'array come parametro. Questo esempio permette di evidenziare un aspetto molto importante: proprio per il fatto che un array è una zona di memoria allocata "al volo", i singoli elementi contengono all'inizio i dati che erano precedentemente presenti in tale area, per cui quella di inizializzare il valore dell'array come nell'esempio (o almeno chiedersi se sia il caso di farlo) è una buona pratica che permette di evitare errori inaspettati ed apparentemente inspiegabili.
Altre conseguenze di quanto detto fino ad ora sono le seguenti:

1) scrivere `*variabile` (contenuto della locazione puntata) equivale a scrivere `variabile[0]`;
2) nel caso in cui occorra passare un puntatore ai vari elementi dell'array (per esempio per la funzione `scanf`), anziché scrivere `&(nomearray[i])`, si può scrivere semplicemente `nomearray + i`;

3) è possibile creare dei “sotto-array” semplicemente assegnando ad un puntatore l’indirizzo di un certo elemento dell’array: `sottoarray = array + i`. Il primo elemento di `sottoarray` sarà l’i-esimo elemento di `array`. Questo particolare risulta molto utile nelle funzioni che impiegano tecniche di ricorsione;
4) disponendo di un puntatore ad un’area di memoria, è possibile utilizzare quell’area come se fosse un array, per leggerne o scriverne il contenuto.

Per iniziare un array a copiare un array in un altro, si può usare un ciclo `for`, oppure si possono usare le funzioni che sono disponibili a questo scopo nella libreria *mem.h*. Le funzioni più utili sono la funzione `memset`, che serve per inizializzare un’area di memoria con un valore particolare, la funzione `memmove` (o `memcpy`), che copia il contenuto di un blocco di memoria in un altro e la funzione `memcmp`, che confronta due blocchi di memoria. La sintassi delle funzioni è la seguente:

```
void *memset(void *s, int c, size_t n);
void *memmove(void *dest, void *src, size_t n);
int memcmp(void *s1, void *s2, size_t n);
```

La prima prende come parametro il puntatore all’area di memoria `s`, il valore `c` con cui inizializzare l’area e la lunghezza in byte `n` dell’area stessa. La seconda richiede il puntatore all’area di destinazione `dest`, a quella sorgente `src` e la lunghezza da copiare `n` (in byte). La terza prende i puntatori alle due aree e la lunghezza (rispettivamente `s1` e `s2` ed `n)` e restituisce il valore 0 solo se le due aree sono uguali. Un’altra funzione utile è `memchr`, che permette di ricercare un carattere in un blocco di memoria:

```
void *memchr(void *s, int c, size_t n);
```

La funzione richiede il puntatore al blocco di memoria, il carattere da ricercare e restituisce il puntatore alla locazione che contiene il primo carattere trovato, oppure `NULL` in caso di nessuna occorrenza trovata.
Si ricorda che anche le stringhe sono degli array! Questo implica che quanto detto a proposito dei puntatori si applica anche a loro ed in alcuni casi è possibile utilizzare le funzioni dedicate alla manipolazione degli array anche per le stringhe e viceversa, basta prestare attenzione al fatto che le stringhe devono sempre terminare con il carattere 0x00.

10. Struttura e leggibilità del codice

INTRODUZIONE

Una delle maggiori potenzialità del linguaggio C risiede nella sua grandissima versatilità ed espressività. Esso permette cioè di descrivere un determinato comportamento in modi diversi e comunque sempre in maniera molto compatta ed efficiente. Questa caratteristica, di per se positiva, può potenzialmente essere fonte di diversi problemi se non viene ben gestita! Infatti un codice molto compatto, se non ben progettato e scritto può risultare poco leggibile, scarsamente testabile e documentabile, poco portabile e difficilmente aggiornabile. Al contrario queste caratteristiche dovrebbero sempre essere considerate importanti almeno quanto il corretto funzionamento del programma stesso. Per facilitare questo compito il linguaggio C mette anche a disposizione alcuni strumenti, che gestiti con cura dal programmatore possono assicurare ottimi risultati. In questo capitolo verranno analizzati alcuni tra questi strumenti, nonché alcune regole pratiche da seguire per potere scrivere dei buoni programmi.

LEGGIBILITÀ DEL CODICE

Si provi a capire la funzione svolta dal codice C che segue:

```
char *dtb(char n) {
int i; char s[8];
s[8]=0;
for(i=0; i<8; i++)
s[i]=48+(n>>i)&1;
return s;
}
```

Si ritenti ora con il seguente:

```
/* Conversione decimale -> binario
   Parametri:
   - dec = numero intero (0-255)
```

```
    Restituisce:
    - "dec" in binario (stringa) */
char *DecToBin(unsigned char dec)
{
  int i;  /* indice posizione */
  char bin[9];  /* stringa */

  bin[8]=0x00; /* terminatore stringa */

  for(i=0; i<8; i++)
  {
    /* scorre n a destra di i posti
       e considera solo l'LSB
       (equivale a: n/(2^i) AND 1) */
    bin[i] = '0' + (dec>>i)&1;
  }

  return bin;
}
```

Le due routine eseguono esattamente le stesse operazioni, ma nel secondo caso non c'è niente da aggiungere, il codice è auto-esplicante. Cosa rende il secondo codice molto più comprensibile del primo? In primo luogo i commenti che in C possono essere costituiti da testo qualsiasi, anche su più linee, racchiuso tra i simboli /* e */. Non è raro per chi è alle prime esperienze con la programmazione, rileggere il codice scritto solo una settimana prima e trovarlo assolutamente incomprensibile! Esperienze come queste insegnano quanto sia importante un uso massiccio di commenti nei propri programmi, anche quando si è convinti di poterne fare a meno. Un altro accorgimento per rendere il codice più leggibile è quello di utilizzare *l'indentazione*.
L'indentazione consiste nell'utilizzare spazi (o tabulazioni) in modo da allineare verticalmente il codice che appartiene ad uno stesso blocco o contesto. Ad esempio nel programma precedente tutto il codice dentro la funzione inizia dopo due spazi, così come quello dentro al ciclo `for` (che si troverà quindi a circa quattro spazi dal bordo della pagina). Questo accorgimento permette di suddividere logicamente e semanticamente il codice quindi di poterlo leggere e seguire più facilmente. L'indentazione è anche molto utile per rendersi conto in maniera veloce di quante parentesi graffe occorre chiudere alla fine di una routine.
Un ulteriore (e forse decisivo) aiuto alla leggibilità del codice è dato dall'utilizzo di nomi significativi per le variabili e le funzioni. È possibile utilizzare anche nomi piuttosto lunghi, composti anche da lettere maiuscole (il C è *case sensitive*!) o contenenti il sottotratto "_" (*underscore*) per evidenziare diverse parole all'interno dello stesso nome: è molto più espressivo un nome come `vettore_coefficienti[]` che non `v[]`. Questa idea si estende al punto da seguire delle precise convenzioni, come quella di utilizzare la prima lettera del nome di una variabile per indicarne il tipo di dato da esso rappresentato, ad esempio una variabile di tipo `float` si potrà chiamare fLunghezza, mentre una variabile intera potrebbe chiamarsi `iContatore`...
Ovviamente applicare scrupolosamente tutti questi accorgimenti comporta un leggero aumento del tempo necessario alla scrittura del codice, però i vantaggi che ne derivano sono talmente grandi da giustificare sempre il maggiore sforzo.

STRUTTURA DEI PROGRAMMI

Così come è bene rispettare certe regole nella scrittura delle singole linee di codice e delle routine, è anche opportuno considerare alcuni accorgimenti nell'organizzazione e nella scrittura dell'intero programma. In questo caso gli obiettivi a cui mirare sono, oltre alla leggibilità, anche la riutilizzabilità del codice, la facilità di modifica o aggiornamento e la portabilità (indipendenza dall'hardware). Prima di vedere in dettaglio come si possono raggiungere questi risultati è il caso di introdurre i "file header". Si tratta di file che hanno lo stesso nome dei file di programma, ma hanno estensione ".h" anziché ".c". Mentre i file di programma contengono il codice vero e proprio delle routine, negli header sono presenti soltanto i prototipi delle funzioni e la dichiarazione di macro e costanti. Per prototipo di una funzione si intende una dichiarazione contenente il nome della funzione ed i tipi da essa utilizzati in ingresso ed uscita. L'uso dei prototipi non è strettamente indispensabile, però permette di trovare facilmente degli errori poco visibili nella scrittura del codice. Infatti se nel programma viene richiamata una funzione utilizzando dei parametri di un tipo diverso da quello dichiarato nel prototipo, il compilatore darà un avvertimento (*warning*) o addirittura un errore. L'utilizzo dei file header ha lo scopo di separare la dichiarazione delle funzioni (cioè la loro "interfaccia") dal codice vero e proprio. Conoscendo l'interfaccia di una funzione (cioè le variabili in ingresso, quelle in uscita ed i rispettivi tipi) è possibile utilizzarla senza bisogno di conoscere il suo funzionamento interno e la sua implementazione.
Questo approccio nella programmazione viene chiamato *information hiding*, proprio perché tende a nascondere le parti più complesse e quindi meno leggibili del programma. Questo meccanismo che potrebbe sembrare un po' complicato, è lo stesso utilizzato ogni volta che viene usata una funzione di libreria. Anche in quel caso viene richiamto un file header e vengono utilizzate funzioni di cui è nota soltanto l'interfaccia, ma non la reale implementazione. Dovrebbe essere chiaro a questo punto che gli header sono utili sia per aumentare la leggibilità, sia soprattutto per favorire la riutilizzabilità del codice. Infatti è possibile ed in molti casi conveniente, creare delle proprie librerie di funzioni, scrivendo il codice in un apposito file e creando il rispettivo header. Ogni volta che verranno utilizzare quelle funzioni in un programma sarà sufficiente solamente includere il file header!
Ecco un esempio relativo al codice di un'ipotetica libreria che implementa due di funzioni statistiche. Il file *stat.c* conterrà il seguente codice:

```
/* *** File stat.c *** */

#include "stat.h"

/* Funzione Sum */
int Sum(char *vettore, int num_elem)
{
  int i, somma=0;

  for(i=0; i<num_elem; i++)
    somma += vettore[i];

  return somma;
}
```

```
/* Funzione Med  */
int Med(char *vettore, int num_elem)
{
  int i, media=0;

  for(i=0; i<num_elem; i++)
    media += vettore[i];

  return media/num_elem;
}
```

Il file *stat.h* invece conterrà le seguenti dichiarazioni:

```
/* *** File stat.h *** */


/* *** Sum ***
   Restituisce la somma degli elementi
   dell'array di byte "vettore" */
int Sum(char *vettore, int num_elem);


/* *** Med ***
   Restituisce la media degli elementi
   dell'array di byte "vettore" */
int Med(char *vettore, int num_elem);
```

Per utilizzare le due funzioni nei propri programmi basta includere il file *stat.h* (i due file devono risiedere in una directory nota al compilatore, altrimenti occorre specificarla nella direttiva `#include`).
Si nota che il file *stat.c* non ha un `main` proprio perché il codice non deve funzionare da solo, ma deve essere sempre richiamato da un altro programma.

Negli header di solito vengono dichiarate anche le costanti ed i tipi definiti dall'utente, che poi verranno usate nel codice. Il vantaggio in questo caso risiede nel fatto che se il valore di queste costanti deve essere cambiato, è sufficiente farlo una sola volta modificando il file header, senza bisogno di accedere al codice.
Questo offre un modo per parametrizzare i propri programmi e poterli personalizzare a seconda delle esigenze. Alcune costanti tipicamente dichiarate in questo modo sono quelle legate alle dimensioni massime degli array.
Anche per i tipi valgono le stesse considerazioni. Nell'esempio precedente si sarebbe potuto utilizzare il tipo `INTERO_CORTO` ed `INTERO_LUNGO` al posto di `char` ed `int` e definirli così nel file header nel seguente modo:

```
typedef char INTERO_CORTO;
typedef int INTERO_LUNGO;
```

in questo modo le funzioni avrebbero potuto utilizzare anche vettori di `long` o `float`, semplicemente ritoccando queste due righe dell'header!

ESEMPIO PRATICO

Di seguito è riportato un esempio di programma completo che segue tutti gli accorgimenti citati. Nonostante la sua estrema semplicità (dovuta ad esigenze "didattiche"), esso ha una struttura completa e molto simile a quella che potrebbe avere un programma molto più grande e complesso. Il programma dovrebbe essere eseguito su un sistema a microcontrollore che ha lo scopo di tenere sotto controllo i dati provenienti da alcuni sensori, controllando che essi si mantengano entro dei valori limite stabiliti. La condizione di normalità o di avaria è visualizzata su un display LCD alfanumerico. Il programma è costituito da 3 moduli: il modulo *main.c*, che contiene la routine principale del programma, il modulo *sensori.c* che si occupa della lettura dei dati dai sensori ed il modulo *display.c* che invece ingloba le funzioni per il pilotaggio del display. Ciascuno dei moduli ha il proprio header. Va notato che questa struttura rende il programma abbastanza indipendente dall'hardware, infatti esso potrebbe perfino funzionare in maniera "simulata" su un PC, modificando soltanto le funzioni dei moduli *sensori.c* e *display.c* in modo che utilizzino ad esempio `scanf` e `printf` per leggere i dati e visualizzare l'output.
Anche i parametri di funzionamento (limiti impostati e coefficienti di conversione) risultano facilmente modificabili dagli header.

File "main.c"

```
/* ****************************
   *   PROGRAMMA DI CONTROLLO   *
   *     Modulo principale      *
   **************************** */

#include "main.h"


main()
{
  int valore, err_temp, err_press;

  /* ciclo infinito */
  while(1)
  {

    /* Controllo temp. acqua */
    err_temp=0;
    valore=TempAcqua();
    if (valore>MAX_TEMP_ACQUA)
      err_temp=1;

    /* Controllo press. olio */
    err_press=0;
    valore=PressOlio();
    if ((valore<MIN_PRESS_OLIO)||
  (valore>MAX_PRESS_OLIO))
      err_press=1;
```

```
    if ((err_temp==1)||(err_press==1))
    {
      Scrivi("Rilavata anomalia!");
      Buzzer();
    }
    else
      Scrivi("Funzionamento normale");
  }
}
```

File “main.h”

```
/* ****************************
   *   PROGRAMMA DI CONTROLLO   *
   *  Header modulo principale  *
   **************************** */

#include "sensori.h"
#include "display.h"


/* --- Definizione parametri --- */

/* Massima temperatura acqua */
#define MAX_TEMP_ACQUA 98

/* Minima pressione olio */
#define MIN_PRESS_OLIO 8

/* Massima pressione olio */
#define MAX_PRESS_OLIO 21
```

File “sensori.c”

```
/* ***************************
   * Modulo gestione sensori *
   *        - codice -       *
   *************************** */

#include "sensori.h"


/* --------------------------- */
int TempAcqua(void)
{
  int i;

  /* Istruzioni dipendenti
     dall'hardware */
```

```
  return i*COEFF_TEMP;
}


/* -------------------------- */
int PressOlio(void)
{
   int i;

  /* Istruzioni dipendenti
     dall'hardware */

   return i*COEFF_PRESS;
}
```

File "sensori.h"

```
/* ************************
   * Modulo gestione sensori *
   *        - header -        *
   ************************ */


/* -- Definizioni parametri -- */

/* Coeff. di convers. temp. */
#define COEFF_TEMP 1.5

/* Coeff. di convers. press. */
#define COEFF_PRESS 3.2

/* - Prototipi delle funzioni - */

/* Legge il valore del sensore
   della temperatura dell'acqua */
int TempAcqua(void);

/* Legge il valore del sensore
   della pressione dell'olio */
int PressOlio(void);
```

File "display.c"

```
/* ************************
   * Modulo gestione display *
   *        - codice -        *
   ************************ */

#include "display.h"
```

```
/* --------------------------- */
void Scrivi(char *testo)
{
  /* Istruzioni dipendenti
     dall'hardware */
}


/* --------------------------- */
void Buzzer(void)
{
  /* Istruzioni dipendenti
     dall'hardware */
}
```

File "display.h"

```
/* ***************************
   * Modulo gestione display *
   *        - header -       *
   *************************** */

#include <stdio.h>


/* - Prototipi delle funzioni - */

/* Scrive un rigo di
   testo sul display */
void Scrivi(char *testo);

/* Emette un beep */
void Buzzer(void);
```

11. L'uso del preprocessore C

INTRODUZIONE

Nella stesura di programmi di una certa complessità risulta inevitabilmente difficile apportare modifiche coerenti nei vari moduli. Questa necessità può sorgere sia nel caso in cui si vogliono effettuare temporaneamente delle prove o apportare modifiche al programma (per esempio per eseguire un porting da PC a microcontrollore), oppure anche (e forse soprattutto) nella fase di debug, in cui capita di dovere aggiungere o commentare temporaneamente diverse sezioni del codice per isolare gli errori o verificarne il corretto funzionamento. Senza l'uso di strumenti adatti queste operazioni possono comportare inevitabilmente la generazione di una serie di errori involontari ed in alcuni casi molto dannosi in termini di tempo di sviluppo richiesto per la correzione. Può capitare ad esempio (e non è per niente infrequente!) di dimenticarsi di quali modifiche sono state fatte per eseguire dei test, oppure di non eseguire le correzioni dovute su tutti i moduli o anche fare confusione con le diverse versioni dei file. Esistono degli strumenti adatti ad evitare o risolvere questi problemi? La risposta è sì. E la buona notizia è che questi strumenti sono parte integrante del linguaggio C! La parte del compilatore che si occupa di questi aspetti si chiama preprocessore. In realtà il preprocessore è concettualmente un tool separato rispetto al compilatore vero e proprio, esso infatti interviene sul codice prima della fase di compilazione ed ha lo scopo di manipolarlo, dal punto di vista "fisico" si potrebbe dire, intervenendo sul testo in base alle direttive fornite dall'utente. Questo capitolo analizzerà le più comuni direttive per capirne le potenzialità e gli usi.

#INCLUDE

La direttiva `#include` ha lo scopo di includere un file nel processo di compilazione. Ci sono però alcuni aspetti non molto conosciuti riguardo a questa direttiva, che vale la pena di approfondire. Le direttive hanno particolari caratteristiche, quali:

1) ogni direttiva inizia con il carattere #, che deve essere il primo carattere della riga (a parte eventuali spazi);
2) ogni direttiva deve essere scritta su un'unica linea di codice;
3) non è presente il punto e virgola alla fine della riga, questo perché le direttive non sono linee di codice C, ma vengono interpretate (e rimosse) dal preprocessore. Nel caso della direttiva `#include` il file incluso ad esempio viene fisicamente copiato al suo posto.

La direttiva `#include` prevede due sintassi diverse:

```
#include <nomefile.h>
#include "nomefile.h"
```

Entrambe le varianti sono corrette, la differenza nell'uso delle parentesi acute (segni di maggiore e minore) o delle virgolette influenza soltanto l'algoritmo di ricerca del file da includere. Questa caratteristica dipende dalla particolare implementazione (cioè dal particolare compilatore utilizzato), in genere l'uso delle parentesi acute implica che il file venga cercato nella directory dove risiedono le librerie del compilatore, in caso contrario il file viene ricercato anche in directory esterne, come quella che contiene il progetto o nei percorsi definiti dal sistema operativo.
Esiste un'altra interessante variante:

```
#include NOMEFILE
```

dove `NOMEFILE` è una macro definita in altri punti o moduli del programma con la direttiva `#include`. Questa opzione risulta utile quando occorre includere un file diverso a seconda del tipo di codice che si vuole produrre. La macro può essere definita in un header di configurazione, richiamato da tutti i moduli, in cui si specificano anche questi parametri oltre a quelli relativi al programma stesso. Questo accorgimento può essere utilizzato per specificare la macchina o il sistema operativo per cui verrà compilato il programma: `NOMEFILE` potrebbe essere definito nel file di configurazione e corrispondere a *windows.h* o *dos.h*…
Questa tecnica è un primo passo per mantenere la coerenza tra i vari moduli di un programma e può essere ampliata ed automatizzata nel caso della compilazione condizionale.
Se un file header di un programma è richiamato da più moduli diversi si verifica un errore, perché molti identificativi risultano definiti più volte. Questo problema verrà affrontato più avanti nella trattazione.

#DEFINE E #UNDEF

Come già visto la direttiva `#define` permette di definire le "macro", gli identificativi utilizzabili al posto dei frammenti di codice. Il preprocessore letteralmente sostituisce ogni occorrenza di una macro con il frammento di codice (più in generale testo) corrispondente precedentemente definito. La sintassi prevede di utilizzare dopo la direttiva un identificativo e il testo da sostituire, separati da uno o più spazi.
Il testo considerato come corpo della macro è quello compreso tra lo spazio dopo l'identificativo e il termine della linea, quindi nel testo possono essere presenti anche spazi. Se il testo è costituito da un frammento di codice C, è bene evitare di mettere il punto e virgola alla fine, altrimenti nella sostituzione si potrebbero verificare involontarie "interruzioni" di istruzioni o errori simili.
Una caratteristica interessante è che la direttiva `#define` permette di definire macro con parametri, simili quindi a funzioni. Ad esempio:

```
#define POW2(n) 1<<((int) n)
```

oppure

```
#define Swap(t,a,b) {t temp; temp=a; a=b; b=temp;}
```

Nel primo caso la macro viene utilizzata per generare potenze intere di 2, dato l'esponente `n` (le potenze sono calcolate scorrendo 1 a sinistra di 'n' posizioni). Notare la conversione a `int`, che impedisce che il compilatore dia errori se si usa un `float` come parametro (i `float` sono incompatibili con gli operatori bitwise).
Nel secondo esempio invece è stata definita una macro che scambia il contenuto di due variabili. Notare che per fare questa operazione è necessario creare una terza variabile temporanea quindi è necessario fornire tra i parametri il tipo delle due variabili. Il codice C che implementa la funzione `Swap` è racchiuso tra parentesi graffe (quindi si tratta di un "blocco" di codice), proprio perché è necessario definire una nuova variabile indipendente dal contesto. Tutto il codice è stato scritto su una sola linea. Nel caso questo non sia possibile o conveniente, si può terminare la linea col carattere di concatenazione di linea '\', come mostrato di seguito:

```
#define Swap(t,a,b) \
         {t temp; \
          temp = a; \
          a = b; \
          b = temp;}
```

L'uso dei parametri nelle macro può generare molto facilmente errori. Infatti i parametri non sono passati per "valore" come nelle funzioni, ma viene semplicemente copiato il testo usato come parametro. Si consideri ad esempio l'invocazione della macro con parametri come `Swap(int, var+1, c)`, oppure `Swap(int, i, d++)`: l'istruzione "`temp = a`" sarebbe sostituita da "`var+1 = a`" in cui cioè il valore di `a` è assegnato ad 1, non alla variabile `var`, oppure nel caso di `d++` si avrebbero degli incrementi in diversi punti o degli errori di sintassi. Questi problemi possono in parte essere evitati evitare usando le parentesi per racchiudere gli argomenti. Quanto definito con la direttiva `#define` può anche essere rimosso grazie alla direttiva `#undef`.

```
#undef NOME_MACRO
```

Da questo punto del codice in poi sarà possibile usare quello stesso identificativo per indicare una funzione o per redefinire un'altra macro.

#IFDEF E #IFNDEF

Per verificare se un identificativo è già stato utilizzato o meno, si possono utilizzare le due direttive `#ifdef` e `#ifndef`, che servono proprio a considerare valido per la compilazione un tratto di codice se la rispettiva condizione è verificata. Ad esempio, per verificare che non ci siano delle redefinizioni di una determinata macro (che porterebbero ad un errore), oppure per definire in modo differente una macro, si può usare la seguente sintassi:

```
#ifndef POW2
  #define POW2(n) 1<<((int) n)
#endif
```

oppure

```
#ifdef POW2
  #undef POW2
  #define POW2(n) 1<<((int) n)
#endif
```

Questo stesso meccanismo viene usato per risolvere il problema dell'inclusione di uno stesso file header da parte di diversi moduli. L'accorgimento consiste nell'inserire all'inizio ed alla fine di ogni file header le seguenti righe:

```
#ifndef _NomeHeader_H
#define _NomeHeader_H
…
/* codice dell'header */
…
#endif
```

In questo modo, quando l'header viene richiamato la prima volta la direttiva `#ifndef` risulta verificata, perché l'identificativo `_NomeHeader_H` non è ancora stato definito. Il codice seguente viene considerato valido quindi compilato. Lo strano nome della macro è dovuto al fatto che l'identificativo deve risultare unico. Il codice contiene l'intero corpo dell'header e provvede anche a definire l'identificativo `_NomeHeader_H` con una direttiva `#define`. Le successive volte che l'header verrà richiamato l'identificativo risulterà già definito, quindi il codice seguente verrà ignorato. Grazie a questo accorgimento non sarà necessario prestare particolare cura a quante volte ed in quali moduli ciascun header è richiamato.

#IF, #ELIF, #ELSE ED #ENDIF

Queste direttive funzionano in maniera simile a `#ifdef` o `#ifndef`, anzi in alcuni casi possono anche essere utilizzate congiuntamente. Il loro scopo è quello di permettere la "compilazione condizionale", cioè fare in modo che vengano compilati alcuni tratti di codice piuttosto che altri, in dipendenza di alcune condizioni specificate dall'utente.
La sintassi generale è la seguente, anche se sono possibili diverse variazioni e combinazioni (le sezioni `#elif` ed `#else` sono opzionali):

```
#if espressione-1
  <sezione di codice 1>
#elif espressione-2
  <sezione di codice 2>
  .
  .
#elif espressione-n
  <sezione di codice n>
#else
  <codice di default>
#endif
```

Viene considerato valido soltanto il codice corrispondente all'espressione che risulta verificata. Se nessuna condizione è verificata viene eseguita quello presente nella sezione `#else`. Le condizioni sono valutate in modo da dare un risultato intero, se il risultato è 0 la condizione non è verificata. Un esempio di utilizzo di queste direttive è quello legato alla compilazione automatica di codice per più macchine diverse, come accennato in precedenza. Uno o più moduli potrebbero contenere le seguenti istruzioni:

```
#if SYSTEM == DOS
  #define HDR "dos.h"
#elif SYSTEM == Z80
  #define HDR "z80.h"
#elif SYSTEM == PIC
  #define HDR "pic.h"
#else
  #define HDR "generic.h"
#endif

#include HDR
```

In questo caso le direttive condizionali vengono utilizzate per definire il nome di un file header da importare che contiene dei parametri relativi ad un particolare tipo di macchina. Il valore della macro `HDR` viene definito uguale al nome di uno dei file header da includere (con la direttiva `#include`). Il valore della macro `SYSTEM` è assegnato in un file header apposito, che ha proprio lo scopo di configurare i parametri di compilazione e in cui sono definiti anche le macro `DOS`, `Z80` e `PIC`:

```
#define DOS
#define Z80
#define PIC

#define SYSTEM DOS
```

Lo stesso sistema si può utilizzare anche per abilitare o disabilitare parametricamente alcune funzioni o per selezionare tra differenti tipi di implementazioni disponibili (ottimizzate per la massima velocità, la minima occupazione di memoria, etc...). Un esempio può essere il seguente, in cui grazie all'uso delle direttive condizionali si può selezionare tra un operazione eseguita con maggiore precisione (in virgola mobile), ma molto più lentamente ed un'altra un po' meno precisa ma molto più veloce (eseguita in aritmetica intera):

```
#if USE_FLOAT
  typedef float TIPO_DATI;
#else
  typedef int TIPO_DATI;
#endif
…
TIPO_DATI c,r;

#if USE_FLOAT
  /* versione float */
```

```
  c = 2*3.14159*r;
#else
  /* versione intera */
  c = (1608*r)>>8;
#endif
```

In un header sarà definito `USE_FLOAT` col valore 0 o 1:

```
#define USE_FLOAT 1
```

La possibilità di parametrizzare un intero programma in maniera così semplice e veloce è sicuramente una possibilità molto interessante (per maggiori dettagli sull'uso dell'aritmetica intera invece si rimanda al capitolo 14). Le possibilità offerte dalla compilazione condizionale non si esauriscono qui, una delle più interessanti infatti è quella legata al debug dei programmi. È molto comune inserire in questa fase diverse righe di codice che hanno il solo scopo di verificare se il programma sta eseguendo le operazioni che ci si aspetta o che visualizzano su qualche dispositivo di output il contenuto di variabili critiche o addirittura una specie di log dell'esecuzione del programma stesso. Terminata la fase di debug di solito risulta molto scomodo dovere scorrere l'intero codice per rimuovere queste istruzioni e spesso anche una banale dimenticanza può portare ad un degrado inaspettato delle prestazioni. Utilizzando le direttive di compilazione condizionale è possibile non solo disabilitare tutte queste parti di codice simultaneamente, ma anche potere selezionare facilmente tra più livelli di debug (verifica semplice, stampa del contenuto di variabili, log dell'esecuzione…). Un esempio può essere il seguente:

```
#include <math.h>
…

float Angolo(float x, float y)
{
  float m;

  #if DEBUG_VERIFICA
    if (x==0)
      printf("Errore in funz. Angolo!\n");
  #elif DEBUG_STAMPA
      printf("x=%f – y=%f\n", x, y);
  #endif

  m=y/x;
  return atan(m);
}
```

La funzione calcola un angolo a partire da due coordinate rettangolari, usando la funzione arcotangente. Dal momento che per fare questo è necessario eseguire una divisione, può capitare che il denominatore in qualche caso sia uguale a zero, causando un errore. Le due funzioni di debug servono proprio per controllare che non si verifichi questa condizione. Per attivarle basta inserire la seguente riga di codice:

```
#define DEBUG_STAMPA 1
```

#PRAGMA

La direttiva `#pragma` è un po' particolare in quanto non esegue nessuna operazione, ne serve a richiamare altre direttive o funzioni esclusive dei tool di sviluppo che si stanno utilizzando. Questo fa sì che sia possibile utilizzare direttive del preprocessore aggiuntive o impostare dall'interno del programma una serie di parametri che di solito si settano nella linea di comando, oppure attivare alcune funzioni che verranno utilizzate nella fase di linking. Si tratta nella maggior parte di casi di funzioni non standard e proprio per questo motivo ogni preprocessore si limita ad ignorare le direttive `#pragma` che non riconosce, senza dare errori. Giusto per fare alcuni esempi, il compilatore SDCC, ha tra le direttive `#pragma` le seguenti:

```
#pragma less_pedantic
#pragma nogsce
#pragma noinv
```

La prima evita che il compilatore segnali gli errori banali di sintassi, la seconda e la terza disabilitano due delle ottimizzazioni del compilatore (l'eliminazione delle sottoespressioni comuni e l'inversione del loop).

12. Strutture dati dinamiche

INTRODUZIONE

Esistono molte situazioni in cui è necessario gestire dati la cui dimensione non è noto a priori o può addirittura variare durante l'esecuzione del programma. Si pensi ad esempio alla memorizzazione di una rubrica telefonica, ad un elenco dei nomi dei file presenti in una directory (con tanto di lunghezza ed attributi vari), ad una lista di compiti da svolgere che vengono generati e messi in coda prima di essere eseguiti. A prima vista una situazione del genere potrebbe sembrare difficilmente gestibile in C, dal momento che il linguaggio impone un dimensionamento preciso e definitivo dei dati utilizzati. Una prima soluzione banale, ma assolutamente non ottimizzata, potrebbe essere quella di creare oggetti (ad esempio array) di dimensioni più grandi di quelle effettivamente necessarie. Questa soluzione comporta un notevole spreco di memoria, è poco flessibile e non dà garanzie assolute di funzionamento in tutte le condizioni.
La soluzione migliore è quindi allocare la memoria per ogni elemento solo quando esso deve essere creato e deallocarla nel caso in cui l'elemento viene cancellato. Questo può essere fatto con le funzioni `malloc` e `free` della libreria *stdlib.h*, già descritte ed utilizzate nei precedenti capitoli. In questo modo però sarebbe necessario utilizzare una variabile per ogni nuovo elemento, ricadendo nei problemi di disponibilità di risorse. Per potere gestire in maniera efficiente tutti gli elementi creati è necessario quindi "collegarli" in qualche modo, formando così una struttura unitaria, che fa capo ad una sola variabile. Questo, è possibile utilizzando i puntatori. Strutture dinamiche di questo tipo sono utilizzate in molti campi, ad esempio nella gestione di database, in alcune interfacce hardware (FIFO), in alcuni formati di file ed in molti file system (FAT32, Unix…).

LE LISTE

La lista è la struttura dinamica più semplice e intuitiva ed è anche la più utilizzata. Una lista è simile ad un array, i cui elementi possono essere però aggiunti e cancellati singolarmente. Ciascun elemento è collegato al successivo, utilizzando apposite informazioni contenute in esso stesso. In questo modo è sufficiente conoscere il primo elemento (la "testa" della lista) per potere gestire l'intera struttura (figura 12.1).
Più in dettaglio, ogni elemento della lista è costituito da una struttura con (almeno) due campi: il primo è il dato vero e proprio (che può essere indifferentemente un tipo pre-

definito, un array, una stringa, una struttura…), il secondo è un puntatore all'elemento successivo. Una struttura adatta ad implementare una lista potrebbe essere la seguente:

```
typedef struct Elm {
  int dato;
  struct Elm *prossimo;
} Elemento;
```

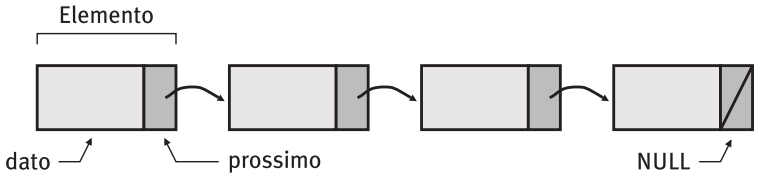


Figura 12.1
Rappresentazione schematica di una lista

Nell'esempio è stato definito un tipo strutturato composto da due campi: un dato, in questo caso di tipo intero ed un puntatore alla stessa struttura.
Nel programma verrà dichiarato soltanto un elemento, che farà capo ad una variabile (o ad un puntatore) e che corrisponderà alla testa della lista, gli altri elementi verranno aggiunti di seguito quando sarà necessario. L'ultimo elemento (la "coda" della lista) si distingue per il fatto che il suo campo `prossimo` punta al valore costante `NULL`. Ecco un esempio in cui una lista di due elementi viene costruita "manualmente", passo passo:

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

typedef struct rec {
  int dato;
  struct rec *prox;
} record;


main() {
  record lista, *elem;

  /* primo elemento */
  lista.dato = 76;
  lista.prox = NULL;

  …

  /* secondo elemento */
  elem=(record *) malloc(sizeof(record));
  elem->dato = 8;
  elem->prox = NULL;
```

```
    lista.prox = elem;

    printf("Secondo dato = %d", (lista.prox)->dato);
}
```

Il codice esegue i passi descritti prima: viene allocata una variabile dello stesso tipo degli elementi della lista (`lista`), che costituirà la testa della lista stessa ed un puntatore ad un elemento (`elem`), che verrà utilizzato come variabile temporanea. Il primo elemento viene subito inizializzato (notare l'assegnazione di `NULL` al puntatore, dal momento che non ci sono ancora altri elementi). Il secondo elemento viene creato successivamente, invocando la funzione `malloc`. Notare che per determinare la quantità di byte da allocate è stata usata l'istruzione `sizeof`, che restituisce proprio questa informazione in base al tipo specificato. La funzione `malloc` restituisce un puntatore al nuovo elemento creato. Il puntatore viene prima utilizzato direttamente per inizializzare i campi del nuovo elemento, poi viene copiato nel campo `prox` dell'elemento precedente in modo da creare il collegamento tra i due. Per ricavare il valore del dato del secondo elemento (usato nella `printf`), è stato necessario partire dal primo, procurarsi il valore del puntatore al secondo elemento (`lista.prox`) quindi selezionare il campo dato della struttura puntata.

Aggiungere elementi alla lista

Aggiungere ulteriori elementi alla lista creata nell'esempio precedente è un'operazione meno intuitiva e mette in luce una caratteristica delle liste di questo tipo: non è possibile accedere direttamente all'elemento voluto come accade negli array, ma è sempre necessario partire dal primo elemento e scorrere la lista fino ad individuare quello che interessa! Per aggiungere un elemento in coda infatti è necessario prima di tutto allocarlo, quindi copiare il puntatore ottenuto nel campo prox dell'ultimo elemento della lista (al posto di `NULL`). Per recuperare il puntatore all'ultimo elemento della lista a sua volta è necessario scorrerla tutta (utilizzando tipicamente un ciclo `while`) fino ad incontrare un elemento che ha nel campo `prox` il valore `NULL`. A questo punto è possibile eseguire la copia del nuovo puntatore al suo posto. Questo meccanismo, sebbene apparentemente piuttosto laborioso, si rivela molto flessibile. È possibile ad esempio aggiungere elementi non solo alla fine della lista, ma anche in un punto qualsiasi. Il procedimento è il seguente:

1) si alloca il nuovo elemento;
2) si scorre la lista fino a raggiungere la posizione desiderata (sia n);
3) si copia il campo `prox` dell'elemento n nel campo `prox` del nuovo elemento;
4) si copia al suo posto il puntatore al nuovo elemento.

In figura 12.2 sono mostrate schematicamente queste operazioni.

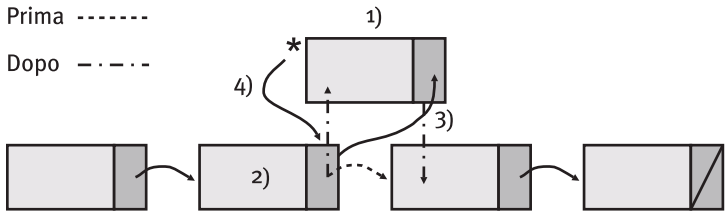


Figura 12.2
Aggiunta di un elemento alla lista

Cancellazione di elementi dalla lista

Per cancellare elementi dalla lista occorre prima di tutto impostare il puntatore dell'elemento precedente in modo che punti a quello a cui puntava l'elemento che si vuole cancellare (scavalcandolo in qualche modo). Successivamente occorre deallocare la memoria occupata da questo, utilizzando la funzione `free`.

L'algoritmo utilizzato è il seguente:

1) scorrere la lista fino all'elemento n-1 (precedente a quello che si vuole cancellare) e conservare il puntatore a questo;
2) recuperare il campo prox dall'elemento n (l'elemento n è puntato dal campo prox di n-1);
3) copiare questo nel campo prox dell'elemento n-1;
4) distruggere l'elemento n con la funzione free.

In questo modo l'elemento n-1 risulterà collegato a n+1 ed n verrà eliminato (figura 12.3). Va notato che questo algoritmo funziona anche nel caso di cancellazione dell'elemento di coda della lista.

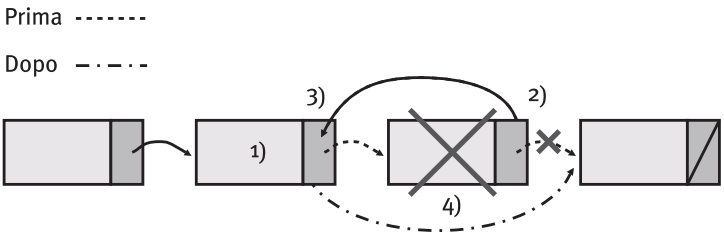


Figura 12.3
Cancellazione di un elemento dalla lista

Altre operazioni

È possibile eseguire anche altre operazioni sugli elementi di una lista, tutte comunque si possono ottenere in maniera simile a quelle già viste. Ad esempio è possibile spostare un elemento da una posizione ad un'altra, eseguendo le operazioni descritte per la cancellazione e l'aggiunta di un elemento (ad eccezione dell'allocazione e della deallocazione). Lo stesso procedimento permette di cambiare la posizione di un intero segmento di lista (consecutivo), in questo caso infatti è sufficiente intervenire soltanto sui puntatori di "raccordo", lasciando inalterati gli altri. Altre operazioni come la cancellazione di pezzi di lista necessitano invece di essere eseguiti sui singoli elementi, dal momento che occorre cancellarli singolarmente dalla memoria.

Liste circolari e bidirezionali

Apportando piccole modifiche alla struttura già vista è possibile ottenere due ulteriori tipi di lista. Per comprenderne meglio l'utilità si immagini di dovere implementare la funzione di selezione di un brano in un lettore MP3: avendo un certo numero di titoli di brani letti da una directory, si dovrà potere scorrere la lista agendo su dei pulsanti. È probabile che ciascun titolo (con altri dati relativi alla lunghezza o alla posizione sul supporto di memoria) sia rappresentato come un elemento di una lista. Scorrendo i titoli in una certa direzione, una volta giunti all'ultimo titolo, normalmente lo scorrimento dovrebbe ricominciare dal primo, ma non solo, dovrebbe anche essere possibile scorrere la lista in entrambe le direzioni. La prima

caratteristica è tipica di una lista *circolare*, la seconda di una lista *bidirezionale*. La prima si può ottenere assegnando al campo `prox` dell'ultimo elemento il puntatore al primo, anziché a `NULL`. In questo modo scorrendo la lista, dopo l'ultimo elemento si ritorna al primo, in modo "circolare". Quando si usano liste circolari (o si rendono circolari liste che prima non lo erano), occorre prestare attenzione ed evitare di usare funzioni che si aspettano un elemento finale con puntatore `NULL`. In mancanza di questo infatti tali funzioni scorrerebbero la lista all'infinito! Il procedimento per rimuovere ed aggiungere elementi in liste circolari è identico al caso di liste normali, l'unica differenza è che non ci si può più riferire ad un inizio, ad una fine o ad una posizione assoluta. L'unico riferimento possibile è quello legato ad un qualche criterio di ordinamento tra i dati.
La lista bidirezionale si ottiene invece utilizzando due puntatori in ciascun elemento, uno collegato all'elemento successivo e l'altro a quello precedente:

```
typedef struct Elm {
  int dato;
  struct Elm *prossimo;
  struct Elm *precedente;
} Elemento;
```

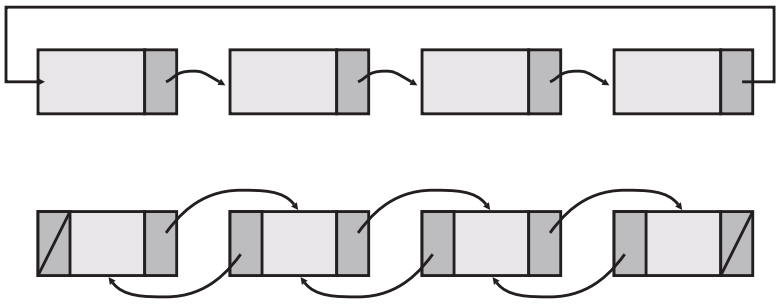


Figura 12.4
Lista circolare (in alto), e bidirezionale (in basso)

Conoscendo un qualsiasi elemento è possibile ottenere sia il successivo che il precedente (nelle liste unidirezionali occorre invece ripartire dall'inizio per ricavare l'elemento precedente!). La gestione di questo tipo di liste è molto simile a quelle unidirezionali, occorre solo compilare correttamente entrambi i campi relativi ai puntatori. Per le liste lineari il campo `precedente` relativo al primo elemento avrà valore `NULL`, così come il campo `prossimo` relativo all'elemento finale. Se si vuole rendere circolare una lista bidirezionale basta collegare i due estremi come spiegato in precedenza. I due tipi di liste sono rappresentati schematicamente in figura 12.4.

Pile e code

Le normali liste, se utilizzate secondo certi criteri, possono implementare altre utili strutture. Tra queste vale la pena di ricordare le pile e le code. Si tratta in entrambi i casi di liste lineari in cui gli elementi vengono aggiunti e prelevati in testa nel caso delle pile, oppure vengono aggiunti in testa e prelevati in coda (o viceversa) nel caso delle code (figura 12.5). Entrambe le strutture vengono utilizzate per accumulare temporaneamente delle informa-

zioni, in attesa che queste vengano utilizzate. Dal momento che la velocità con cui gli elementi vengono aggiunti può essere diversa da quella con cui vengono prelevati, la dimensione delle pile e delle code varia nel tempo. La code vengono utilizzate di solito quando i dati devono essere utilizzati nell'ordine in cui sono stati generati (esempio tipico: un buffer FIFO, oppure un elenco dei compiti da svolgere, in cui quelli generati prima devono essere portati a termine prima). Le pile invece vengono utilizzate in presenza di operazioni "annidate", cioè quando i dati generati dai compiti più interni devono essere consumati per primi proprio da questi (esempio: gestione delle parentesi nelle espressioni aritmetiche o gestione delle variabili create da funzioni annidate).
Le pile e le code possono essere gestite con gli stessi procedimenti visti per le liste lineari.

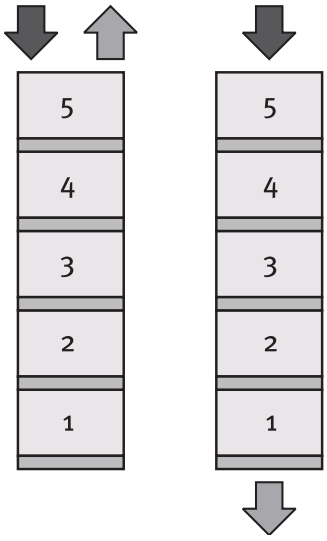


Figura 12.5
Esempio di pila (a sinistra) e coda (a destra)

Gli alberi

In molti casi capita di avere a che fare con dati che hanno una intrinseca organizzazione gerarchica, basti pensare all'insieme dei file contenuti nelle varie sottodirectory di un disco, all'organizzazione di certe reti di telecomunicazione, agli alberi genealogici o all'insieme di possibili scelte che si possono compiere in dipendenza da quelle intraprese prima. Sebbene sia possibile rappresentare questi dati utilizzando strutture lineari (come le liste), spesso è molto più conveniente rappresentarle con una struttura che rispecchia la loro organizzazione, in quanto risulta più semplice l'aggiunta, l'aggiornamento e la ricerca delle informazioni. Le strutture dinamiche utilizzate in questi casi vengono chiamate "alberi" (a causa della loro forma grafica) e sono ottenuti con una tecnica simile a quella utilizzata per le liste (figura 12.6).

Si inizia definendo un tipo strutturato corrispondente ad un elemento dell'albero ("nodo"), costituito da un campo utilizzato per memorizzare i dati e da un certo numero di puntatori che serviranno per collegare i nodi "figli". Un esempio di struttura di questo tipo è il seguente:

```
typedef struct Nd {
  int dato;
```

```
    struct Nd *ramo_sx;
    struct Nd *ramo_ct;
    struct Nd *ramo_dx;
} Nodo;
```

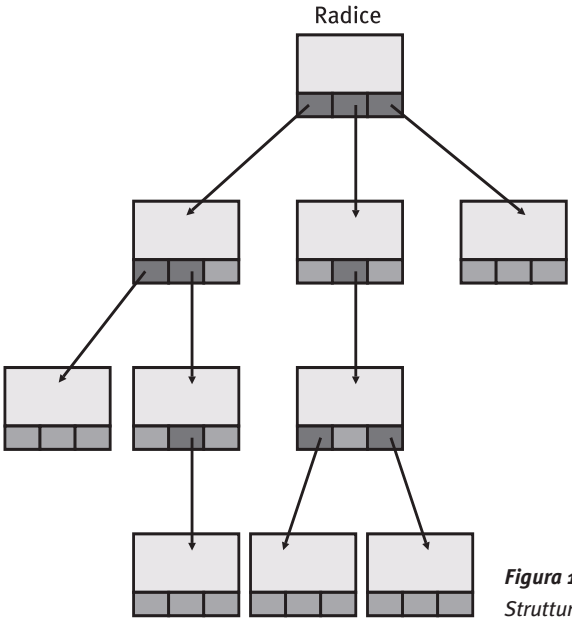


Figura 12.6
Struttura di un albero

In questo caso si è supposto che da ciascun nodo possano partire un massimo di tre rami. Questo ovviamente dipende dalla particolare applicazione. In molti casi pratici per fortuna il numero massimo di rami è conosciuto a priori, semplificando molto la gestione della struttura. Normalmente si parte da un nodo principale chiamato "radice" ed a questo si aggiungono via via gli altri nodi, allocandoli e collegandoli ai rispettivi genitori tramite i puntatori. Ai rami non collegati viene assegnato il valore `NULL` come nel caso delle liste.
Proprio come nel caso delle liste, anche per gli alberi è necessario scorrere gli elementi esistenti prima di potere aggiungere o rimuovere un nodo; il problema in questo caso è che non esiste un solo percorso, ma ne possono esistere moltissimi (in dipendenza dal numero di nodi e dalla forma dell'albero). Questo problema è parzialmente attenuato dal fatto che nelle applicazioni comuni spesso l'albero viene costruito seguendo un certo criterio, ad esempio secondo un certo ordinamento dei dati contenuti nei vari nodi. In base a questo è possibile quindi anche scegliere il ramo da visitare quando occorra aggiungere o cancellare un determinato nodo (il percorso dipende in pratica dai dati in esso contenuto e viceversa, come avviene ad esempio con i nomi dei percorsi dei file e la loro posizione nelle sottodirectory). Quando questa condizione è verificata, l'implementazione e la gestione degli alberi non risulta particolarmente complessa e può essere impiegata anche su piccoli sistemi di calcolo. Invece nel caso in cui non sia possibile applicare queste semplificazioni, la gestione degli alberi risulta alquanto più complessa e richiede l'uso di algoritmi appositi. Ad esempio nel caso in cui occorra scorrere l'intero contenuto dell'albero è possibile utilizzare gli algoritmi di visita in "profondità" o in "ampiezza". Nel primo caso si visita, a partire dalla

radice, un preciso ramo (per esempio quello più a sinistra), quindi lo stesso ramo del nodo figlio e così via, fino a quando non si incontra un puntatore a `NULL`. A questo punto si cambia ramo e si continua nello steso modo. Quando si esauriscono i rami di un certo nodo si ritorna al nodo genitore e si cambia ramo. Questo procedimento porta all'esplorazione dell'albero nel senso della lunghezza. Viceversa la visita in "ampiezza" consiste nel visitare prima tutti i nodi che si trovano allo stesso livello, quindi passare al livello successivo.
Entrambi gli algoritmi risultano abbastanza complessi da implementare, a meno di non fare uso di tecniche ricorsive, che però risultano molto esigenti in termini di risorse di memoria e possono risultare poco efficienti.

ALTRE STRUTTURE

L'idea alla base delle strutture dinamiche può essere estesa ulteriormente, fino alla creazione di strutture con forma e caratteristiche assolutamente arbitrarie. Una delle strutture più generiche che è possibile pensare è il *grafo*. Un grafo (figura 12.7) è un insieme di nodi, collegati appunto in modo arbitrario (questa non è esattamente una definizione rigorosa, ma rende comunque l'idea…). Molte informazioni possono essere pensate come grafi: un insieme di città e le strade che le uniscono, l'insieme di relazioni tra un gruppo di persone, i circuiti elettrici, le relazioni tra un insieme di idee o concetti, le reti di telecomunicazioni, etc. Anche le liste e gli alberi si possono pensare come casi particolari di grafi.

Potere rappresentare e gestire questo tipo di dati può aprire la strada a moltissime applicazioni interessanti. Purtroppo però gli algoritmi di gestione dei grafi possono essere (nel caso

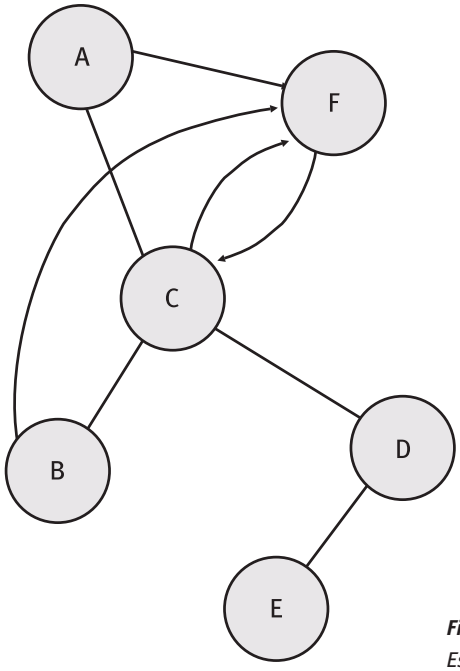


Figura 12.7
Esempio di grafo

generale) abbastanza pesanti da implementare ed eseguire. Il caso più semplice è quello in cui il numero di nodi è costante. In questo caso si può utilizzare una struttura identica a quella vista per gli alberi, con la differenza che i puntatori possono indicare elementi che si trovano in una posizione qualsiasi. Il numero di puntatori da utilizzare nella struttura è noto a priori, infatti al massimo possono esserci tanti collegamenti quanti sono i nodi (eventualmente meno uno). Il problema principale nella gestione dei grafi è costituito dal fatto che spostandosi da un nodo ad un altro per eseguire un algoritmo (calcoli, ricerche…) può capitare di rimanere bloccati in percorsi ciclici. Per evitare questo esistono diversi algoritmi più o meno complessi, uno dei quali consiste banalmente nel "ricordarsi" quali nodi o collegamenti si sono già visitati.

DALLA TEORIA ALLA PRATICA

Per sperimentare l'uso delle strutture dinamiche, di seguito verrà descritto il codice che implementa una delle strutture più utili in pratica: la lista. Purtroppo, come è facile immaginare, la grande flessibilità offerta dalle strutture dinamiche ha un costo aggiuntivo che va attentamente valutato in fase di progettazione. Il "costo" di cui si parla si manifesta principalmente sotto due forme: quantità di risorse di memoria richieste per implementare la struttura e tempo di sviluppo.
La quantità si memoria richiesta per l'implementazione di una struttura dinamica è sempre maggiore di quella richiesta da una equivalente struttura statica, a parità di dati memorizzati. Questo è dovuto al fatto che in una struttura dinamica è necessario memorizzare per ogni dato, uno o più puntatori. L'impatto dei puntatori sulla quantità di memoria totale richiesta può variare a seconda del sistema utilizzato e dalla dimensione dei dati e, in alcuni casi, può essere tutt'altro che trascurabile! Si consideri ad esempio un sistema basato su un microprocessore a 32 bit: è verosimile che ciascun puntatore richieda proprio 32 bit per essere memorizzato. Utilizzando per ciascun elemento della struttura soltanto un dato ad 8 bit, i 4/5 (cioè l'80%) della quantità di memoria richiesta saranno occupati proprio dai puntatori! Perfino su piccoli sistemi ad 8 bit può essere comune avere puntatori a 16 bit, e utilizzando dati ad 8 bit, ben i 2/3 (il 66%) della memoria utilizzata dalla struttura sarà occupata dai puntatori.
È chiaro che in queste condizioni l'utilizzo delle strutture dinamiche non risulta conveniente. Una prima soluzione in questi casi è quella di cercare di "raggruppare" i dati in modo da associarne una quantità maggiore a ciascun elemento della struttura, in questo modo il peso percentuale dei puntatori ed il loro numero risulta considerevolmente ridotto. Questa tecnica è simile a quella utilizzata in alcuni file system per riuscire ad indirizzare in maniera efficiente la grandissima quantità di dati che possono essere contenuti in un hard disk. Anziché indirizzare i singoli byte, vengono indirizzati blocchi o *cluster* grandi decine di KByte.
Nel caso dei file system questo può comportare un certo spreco di risorse (dovuto ai cluster non utilizzati per intero), nel caso delle strutture dinamiche invece questo può essere evitato accodando i dati con criteri più specifici. È possibile anche limitare la quantità di memoria richiesta utilizzando alcune tecniche particolari applicabili a certe strutture dinamiche. Ad esempio nell'implementazione di pile e code è possibile allocare un array sufficientemente grande e memorizzare delle variabili "indice" che tengano conto di quale elemento dell'array costituisce la testa o la coda. Ogni volta che un elemento viene aggiunto o prelevato dall'array il rispettivo indice viene incrementato (o

decrementato). Nel caso delle code l'array deve essere utilizzato in maniera circolare, ovvero incrementando l'indice dalla posizione di coda, questo passa nella posizione di testa. Occorre controllare se, dopo un'operazione, i due indici assumono lo stesso valore: in questo caso la coda è piena o vuota.
Nell'uso di strutture dinamiche è facile commettere errori nella gestione dei puntatori e questi problemi si ripercuotono inevitabilmente sul tempo di sviluppo richiesto dall'applicazione. Alcuni dei problemi più frequenti sono: la scrittura involontaria di aree di memoria utilizzate per altri scopi (causata da un'assegnazione errata di un puntatore), la produzione di "garbage", cioè di elementi riferiti da nessun puntatore e mai deallocati e l'utilizzo di *dangling references*, cioè di puntatori che fanno capo ad aree di memoria già deallocate (quindi contenenti dati non significativi). L'unico modo per evitare questi problemi è prestare molta attenzione nella scrittura del codice e valutare attentamente tutti i messaggi di warning forniti dal compilatore.

IMPLEMENTAZIONE DELLE LISTE

Come già detto le liste sono sicuramente tra le strutture dinamiche più comuni ed utili e possono essere pensate come punto di partenza per l'implementazione di tutte le altre strutture. Per questo motivo di seguito verrà mostrato un esempio di codice relativo all'implementazione delle più comuni funzioni di gestione delle liste. Le funzioni riportate hanno caratteristiche generali e possono essere utilizzare in qualsiasi tipo di applicazione. A scopo esemplificativo si supponga di utilizzare tali funzioni in un programma che implementa una classica rubrica telefonica. Un'applicazione di questo tipo è didatticamente interessante perché, oltre a risultare molto intuitiva, richiede l'utilizzo di tutte le funzioni di gestione delle liste.

La testa della lista

Il modo più generico per gestire una lista è quello di dichiarare un puntatore ad un elemento ed usarlo come variabile di riferimento per puntare alla testa della lista. Nel codice che segue è stata fatta invece una piccola semplificazione allo scopo di rendere le funzioni meno complesse ed evitare l'uso di puntatori doppi (che risulterebbero meno intuitivi). Come testa della lista viene usato un elemento dichiarato appositamente, che non verrà utilizzato per memorizzare i dati, ma solo per puntare al primo elemento utile.
Il tipo utilizzato per gli elementi della lista è il seguente:

```
typedef struct rec {
  TipoDati dato;
  struct rec *prox;
} record;
```

Nel caso della rubrica telefonica il tipo di dati da utilizzare deve contenere almeno un nome ed un numero di telefono, entrambi sotto forma di stringhe. Ad esempio:

```
typedef struct {
  char Nome[32];
  char Tel[16];
} TipoDati;
```

La lista farà capo alla seguente variabile che verrà dichiarata ed inizializzata nel `main` come segue:

```
record lista;
lista.prox=NULL;
```

Aggiungere elementi

Gli elementi possono essere aggiunti alla lista in modi diversi. Il metodo più semplice è l'inserimento in testa. La seguente funzione implementa proprio questa modalità:

```
int aggiungi_in_testa(record *lista, TipoDati d)
{
  record *nuovo;

  nuovo=(record *) malloc(sizeof(record));
  if (nuovo!=NULL) {
    nuovo->dato = d;
    nuovo->prox = lista->prox;
    lista->prox = nuovo;
    return 1;
  }
  return 0;
}
```

La funzione riceve l'indirizzo della variabile utilizzata come testa della lista e una variabile appositamente inizializzata contenente i dati da aggiungere. Viene quindi creato un nuovo elemento utilizzando la funzione `malloc`, in caso di successo il dato `d` viene copiato nella nuova variabile. Infine i puntatori vengono impostati in modo che la testa della lista punti al nuovo elemento ed il nuovo elemento a quello che precedentemente era il primo. La funzione restituisce 1 se l'allocazione è andata a buon fine, oppure 0 in caso contrario.

L'altra modalità usata frequentemente per aggiungere un elemento ad una lista è l'inserimento in coda. La seguente funzione implementa questa modalità:

```
int aggiungi_in_coda(record *lista, TipoDati d)
{
  record *nuovo, *punt;

  nuovo=(record *) malloc(sizeof(record));
  if (nuovo!=NULL) {

    punt=lista;
    while(punt->prox!=NULL)
    punt=punt->prox;

    nuovo->dato = d;
    nuovo->prox = NULL;
    punt->prox = nuovo;
    return 1;
```

```
    }
    return 0;
}
```

La funzione è simile alla precedente: prima alloca un nuovo elemento, quindi lo compila ed infine imposta i puntatori in modo che l'ultimo elemento della lista punti al nuovo e quest'ultimo punti a `NULL`. Notare il ciclo `while` utilizzato per scorrere la lista fino all'ultimo elemento.

L'aggiunta di un elemento in una posizione arbitraria della lista si può realizzare invece con la seguente funzione:

```
int aggiungi_in_posiz(record *lista, int n, TipoDati d)
{
  record *nuovo, *punt;
  int i;

  nuovo=(record *) malloc(sizeof(record));
  if (nuovo!=NULL) {

    punt=lista;
    for(i=0; i<n; i++) {
      if (punt->prox!=NULL)
        punt=punt->prox;
    }

    nuovo->dato = d;
    nuovo->prox = punt->prox;
    punt->prox = nuovo;
    return 1;
  }
  return 0;
}
```

La funzione accetta anche in questo caso l'indirizzo della testa della lista, il dato da aggiungere e la posizione in cui aggiungerlo. La posizione 0 corrisponde ad un inserimento in testa, mentre una posizione uguale o maggiore al numero di elementi presenti nella lista si traduce in un inserimento in coda. Per spostarsi nella posizione voluta in questo caso è stato utilizzato un ciclo `for`, al cui interno viene eseguito un controllo per verificare che non sia già stato raggiunto l'elemento di coda. I puntatori vengono aggiornati come spiegato nel precedente paragrafo.
Le funzioni viste funzionano correttamente anche in caso di lista vuota o lista con un solo elemento. Grazie alla semplificazione adottata all'inizio infatti non è stato necessario distinguere e trattare in maniera differenziata questi (ed altri) casi. In definitiva quindi quello che poteva sembrare uno spreco di memoria (l'allocazione di un elemento non utilizzato), permette effettivamente di ridurre la dimensione del codice.

Cancellare elementi

L'operazione di cancellazione di elementi dalla lista può essere eseguita secondo le stesse modalità viste per l'aggiunta di elementi e anche il codice risulta alquanto simile.

La funzione utilizzata per cancellare l'elemento di testa è la seguente:

```
void cancella_testa(record *lista)
{
  record *temp;

  temp=lista->prox;
  if (temp!=NULL) {
    lista->prox=temp->prox;
    free(temp);
  }
}
```

Il codice è molto semplice: la testa della lista viene fatta puntare al secondo elemento quindi il primo viene distrutto utilizzando la funzione `free`. Con piccole variazioni è possibile modificare la funzione per cancellare l'elemento in coda alla lista:

```
void cancella_coda(record *lista)
{
  record *punt, *prec;

  punt=lista;
  while(punt->prox!=NULL) {
    prec=punt;
    punt=punt->prox;
  }

  free(prec->prox);
  prec->prox = NULL;
}
```

La differenza principale in questo caso è rappresentata dall'uso di un secondo puntatore (`prec`), che viene utilizzato per tenere traccia del penultimo elemento. Questo è necessario perché cancellato l'ultimo, deve essere impostato a NULL proprio il puntatore (campo `prox`) del penultimo.
Anche la procedura di cancellazione dell'elemento n-esimo risulta abbastanza simile a quella vista per l'aggiunta di un elemento nel posto n-esimo:

```
void cancella_posiz(record *lista, int n)
{
  record *punt, *prec;
  int i;

  prec=lista;
  punt=lista->prox;
  for(i=0; i<n; i++) {
    if (punt!=NULL) {
      prec=punt;
      punt=punt->prox;
```

```
    }
  }

  if (punt!=NULL) {
    prec->prox=punt->prox;
    free(punt);
  }
}
```

Anche in questo caso viene utilizzato un ciclo `for` per spostarsi fino all'elemento desiderato e viene utilizzato un puntatore ausiliario per indirizzare l'elemento precedente a quello che si vuole cancellare. L'elemento viene effettivamente cancellato solo se la lista non è già vuota (condizione verificata dall'ultimo `if`).

Leggere gli elementi

Le modalità più comuni per reperire i dati di una lista consistono nel leggere l'elemento di testa, nel leggere quello di coda, quello che si trova in una determinata posizione o eseguire una ricerca. In generale per leggere un elemento è sufficiente scorrere la lista fino alla sua pozione e recuperare il dato in esso contenuto. Per la lettura dell'elemento di testa questa operazione è particolarmente semplice, dal momento che non è necessario lo scorrimento della lista:

```
TipoDati leggi_testa(record *lista)
{
  return (lista->prox)->dato;
}
```

La funzione accetta in ingresso il puntatore alla testa della lista e restituisce il dato ad esso associato. Per la lettura dell'elemento di coda si procede nello stesso modo, ma occorre prima raggiungere la posizione relativa all'ultimo elemento della lista con un ciclo `while`:

```
TipoDati leggi_coda(record *lista)
{
  record *punt;

  punt=lista;
  while(punt->prox!=NULL)
    punt=punt->prox;

  return punt->dato;
}
```

Un ciclo `for` permette di raggiungere una determinata posizione quindi restituire l'elemento corrispondente. Il ciclo utilizzato nella funzione seguente è identico a quello visto in precedenza:

```
TipoDati leggi_posiz(record *lista, int n)
{
  record *punt;
```

```
    int i;

    punt=lista->prox;
    for(i=0; i<n; i++) {
      if (punt->prox!=NULL)
        punt=punt->prox;
    }

    return punt->dato;
}
```

Questo tipo di funzione in genere deve essere personalizzata in base al tipo di dati utilizzati ed i criteri di ricerca da adottare.
Nel programma che gestisce la rubrica telefonica, sarà senz'altro utile immettere un nome (o parte di esso) ed ottenere il numero telefonico corrispondente. In questo caso la funzione di ricerca deve prendere come parametro in ingresso oltre al puntatore alla testa della lista, anche un puntatore ad una stringa e restituire il dato corrispondente o, come in questo caso la posizione della lista corrispondente al dato trovato. La funzione prende in ingresso anche la posizione da cui iniziare la ricerca. Il codice che implementa la funzione è il seguente:

```
int cerca(record *lista, int n, char *stringa)
{
  record *punt;
  int i=0;

  punt=lista->prox;
  while(punt!=NULL) {
    if ((strstr((punt->dato).Nome, stringa)!=NULL)&&(i>=n))
      return i;
    punt=punt->prox;
    i++;
  }

  return -1;
}
```

La funzione non fa altro che scorrere la lista fino alla fine, utilizzando un ciclo `while`, la differenza rispetto ai casi precedenti è che una volta trovata nei dati una stringa nel campo `Nome` contenente la stringa passata in ingresso, la funzione termina restituendo la posizione in cui si è fermata. Se il ciclo raggiunge la coda della lista, la funzione restituirà il valore –1, che indica che nessun elemento soddisfa il criterio di ricerca.
La condizione dell'`if` è verificata se il campo `Nome` associato all'elemento puntato contiene la stringa voluta e (&&) se la sua posizione è maggiore o uguale a quella specificata. La funzione `strstr` è definita nella libreria *string.h* ed il suo compito è quello di ricercare le occorrenze di una determinata sottostringa all'interno di una stringa. Se non viene trovata nessuna occorrenza la funzione restituisce `NULL`.
Come per le altre funzioni la posizione all'interno della lista è numerata a partire da 0 al numero di elementi meno uno.

13. Algoritmi di ricerca ed ordinamento

INTRODUZIONE

In ogni programma esistono alcune funzioni che vengono richiamate più frequentemente di altre nel corso delle elaborazioni, che cioè costituiscono il "nucleo" del programma o di un certo algoritmo. In alcuni casi queste funzioni possono essere richiamate anche più volte all'interno di uno stesso loop. È chiaro che in questi casi il modo in cui le funzioni sono state scritte può incidere notevolmente sui tempi di esecuzione del programma stesso. Questo aspetto, che molto spesso è sottovalutato nell'implementazione di software "applicativo", diventa particolarmente importante in quei casi in cui il programma controlla un sistema che deve necessariamente reagire in tempi brevi o che deve elaborare una grande quantità di dati. Per ottenere funzioni efficienti (dal punto di vista del tempo di esecuzione) si devono considerare due aspetti differenti, ma entrambi molto importanti: l'ottimizzazione del codice e l'utilizzo di algoritmi di complessità computazionale minima. In questo capitolo verrà focalizzato soprattutto il secondo aspetto, applicato ad una classe di funzioni di uso molto generale e per la quale esistono degli importanti risultati anche da un punto di vista teorico. Nei capitoli successivi verrà invece analizzato il primo aspetto, con particolare riferimento alla scrittura di routine matematiche/aritmetiche.

ALGORITMI DI RICERCA

Un algoritmo di ricerca è, come suggerisce il nome, un algoritmo che ha lo scopo di ricercare una particolare sequenza all'interno di un set di dati più o meno grande. Questa esigenza è molto generale e si può presentare in programmi con compiti molto diversi, da applicazioni come i database, a programmi che svolgono elaborazioni di dati a più basso livello, in cui si possono utilizzare delle tabelle di consultazione o di indicizzazione. Normalmente queste tabelle possono essere utilizzate ad esempio per registrare la presenza di determinati dati o per contenere informazioni aggiuntive associate a ciascuno di essi. In molti casi è possibile trattare l'insieme di dati su cui si esegue la ricerca come un array contenente un numero arbitrario di elementi. In questi casi l'implementazione degli algoritmi di ricerca risulta abbastanza semplice e soprattutto veloce, in quanto gli elementi (non necessariamente i dati) sono disposti secondo un preciso ordine ed è possibile accedevi direttamente in base al loro indice.
L'algoritmo di ricerca più semplice è quello di *ricerca lineare* e consiste nello scorrere gli ele-

menti dell'array a partire dal primo, fino ad incontrare (eventualmente) l'elemento ricercato. Il codice seguente implementa l'algoritmo di ricerca lineare in un ipotetico programma che riceve in ingresso un codice binario a 32 bit e fornisce in uscita un secondo codice ricavato da un'apposita tabella di coppie ingresso-uscita. Questo funzionamento è simile a quello utilizzato in alcuni file system per reperire la posizione dei file sul disco, in agenti intelligenti dotati di un comportamento stimolo-risposta ed in molti altri casi.

```
#include <stdio.h>

// *** definizioni ***
#define N_ELEM 128

typedef struct {
  int input;
  int output;
} Dato;

// *** prototipi ***
void inizializza_tabella(Dato *tabella, int n);
int ricerca(int inp, Dato *tabella, int n);

// *** main ***
void main()
{
  Dato tabella[N_ELEM];
  int i, o;

  inizializza_tabella(tabella, N_ELEM);

  printf("Ingresso: ");
  scanf("%d", &i);

  o=ricerca(i, tabella, N_ELEM);

  printf("Output: %d\n", o);
}

// *** Funzione: ricerca lineare ***
int ricerca(int inp, Dato *tabella, int n)
{
  int i;

  for(i=0; ((tabella[i].input!=inp)&&(i<n)); i++){}

  if(i<n)
    return tabella[i].output;  /* Valore trovato */
  else
    return 0;                   /* Valore non trovato */
}
```

Nell'esempio sia gli ingressi che le uscite sono rappresentati da numeri interi a 32 bit, quindi ogni elemento dell'array conterrà una coppia ingresso-uscita. A tal proposito è stato definito il tipo strutturato `Dato` (tramite l'istruzione `typedef`). La tabella è dichiarata all'interno della funzione `main` ed è costituita semplicemente da un array di elementi di tipo `Dato`. Il numero massimo di elementi della tabella in questo caso è fisso, comunque in ogni momento la tabella può anche essere compilata soltanto in parte (ulteriori elementi possono essere aggiunti man mano che questi si rendono disponibili).
La tabella viene inizializzata da una funzione apposita, di cui non è riportato il codice in quanto dipendente dalla particolare applicazione. La funzione di ricerca vera e propria invece viene invocata una volta ottenuto il dato d'ingresso, utilizzando come parametri, oltre all'elemento di input per il quale dovrà essere calcolata l'uscita corrispondente, il puntatore alla tabella ed il numero di elementi contenuti nella tabella stessa. La funzione compie le operazioni descritte prima, che sono implementate semplicemente con un ciclo `for` "modificato" (sarebbe possibile usare anche un più tradizionale `while`). Il ciclo non ripete alcuna operazione, ma ha il solo scopo di incrementare l'indice `i` fino a quando l'ingresso fornito non risulta uguale ad uno degli ingressi memorizzati nella tabella o quando viene superato il numero di elementi memorizzati. A questo punto il ciclo si interrompe conservando il valore dell'indice. Il successivo controllo verifica se è stato effettivamente trovato l'elemento, oppure se il ciclo è arrivato fino alla fine della tabella senza trovare alcuna corrispondenza. In quest'ultimo caso la routine restituisce un valore di default. Per evitare questo ed evidenziare meglio questa condizione occorre restituire un valore apposito ad esempio un numero che sicuramente non è contenuto nei dati ricercati o usare valori negativi per segnalare eventuali errori.
Se il numero di elementi è relativamente piccolo o se è possibile usare una funzione che riesca a convertire valori "sparsi" in una sequenza più o meno ristretta di valori consecutivi (funzione di *hash*) è lecito usare direttamente l'indice della tabella per reperire il valore cercato senza necessariamente eseguire la ricerca.
Riguardo all'algoritmo utilizzato invece è possibile notare che la sua "complessità computazionale" è di ordine N (da qui il nome di "ricerca lineare"). Ciò significa che il tempo richiesto da una ricerca è direttamente proporzionale al numero di elementi presenti nella tabella. Questa è una proprietà della modalità di ricerca adottata, non tanto dell'implementazione: è il problema stesso della ricerca che ha una sua intrinseca complessità minima, che nessun algoritmo consente di aggirare.
Tuttavia, se il problema lo permette, è possibile introdurre degli elementi che rendano la ricerca più facile e veloce.
L'idea di base è che se i dati sono ordinati secondo un certo criterio, è possibile sfruttare proprio il loro ordinamento per trovare più velocemente quello desiderato. Una delle più note tecniche di ricerca basata su questo principio è la *ricerca binaria*. Il procedimento è piuttosto semplice: si supponga di disporre di N dati, in ordine crescente e si consideri il dato in posizione centrale; a seconda che questo sia maggiore o minore di quello cercato è possibile considerare la metà superiore o inferiore quindi si ripete il procedimento in modo iterativo. Per completare la ricerca saranno necessari al massimo $\log_2 N$ operazioni anziché N. Per applicare il metodo della ricerca binaria è necessario preordinare i dati, oppure inserirli direttamente in ordine.

Il codice riportato di seguito mostra come modificare la routine di ricerca vista in precedenza per implementare una ricerca binaria:

```
// *** funzione: ricerca binaria ***
int ricerca(int inp, Dato *tabella, int n)
{
  int i=0;

  do {
    n=n/2;
    if (tabella[i].input>inp)
      i=i-n;
    else
      i=i+n;
  } while ((n>0)&&(tabella[i].input!=inp));

  if (tabella[i].input!=inp)
    return 0;                    // dato non trovato
  else
    return tabella[i].output;    // dato trovato
}
```

La routine scandisce l'array ad intervalli di $n/2^m$ (metà, un quarto, un ottavo…). Questo è ottenuto considerando un indice `i` che inizialmente parte dalla posizione 0, a cui è sommato o sottratto un termine che è ottenuto dividendo iterativamente per 2 la lunghezza dell'array. La somma o la sottrazione sono decisi in base al risultato del confronto tra l'elemento corrente e quello cercato. Ad esempio, si supponga che gli ingressi memorizzati nella tabella siano dei numeri consecutivi compresi tra 0 e 127 e che quello cercato sia l'85. I valori assunti da `n` e da `i` saranno i seguenti:

```
i   n
64, 64
96, 32
80, 16
88, 8
84, 4
86, 2
85, 1
```

Se non si trova l'elemento cercato il ciclo termina quando `n` raggiunge il valore 0. Una cosa importante da notare è che questo metodo funziona bene quando il numero di dati è una potenza di 2 (in questo caso 128). Se questo non è verificato esistono degli accorgimenti per continuare ad utilizzarlo ugualmente, oppure è possibile utilizzare altri metodi simili. Tra questi i più pratici sono quelli che consistono ad esempio nel raggruppare in blocchi i dati in base a delle caratteristiche comuni e conservare l'indice corrispondente a ciascuno di questi blocchi (ad esempio dati relativi a giorni dell'anno potrebbero essere raggruppati per mese o, nel caso di stringhe, per la lettera iniziale).
In questo caso la ricerca richiede al massimo un numero di iterazioni pari alla lunghezza del blocco. Questi algoritmi si rivelano particolarmente utili nel caso di dati organizzati in liste, in cui non è possibile applicare tecniche simili alla ricerca binaria, a causa dell'impossibilità di leggere direttamente un elemento a partire da un indice (la lista deve sempre essere percorsa dall'inizio).

ALGORITMI DI ORDINAMENTO

Una seconda importante classe di algoritmi è rappresentata dagli algoritmi di ordinamento. La loro importanza deriva dal fatto che oltre ad essere impiegati in molte applicazioni comuni, sono spesso utilizzati in combinazione con gli algoritmi di ricerca per aumentarne l'efficienza. Il concetto di "ordinamento" deve essere inteso in senso molto generale: esso non è limitato al caso di ordine numerico o alfabetico, ma può estendersi a casi del tutto arbitrari. Si pensi ad esempio agli algoritmi di scheduling utilizzati dai sistemi operativi per eseguire in time-sharing i diversi processi o per gestire l'accesso alle risorse condivise: la scelta di una particolare sequenza implica necessariamente un ordinamento, che deve tenere conto di fattori come priorità, stato attuale, tempi di attesa, timeout, etc. Escludendo l'operatore di ordinamento, gli algoritmi risultano comunque indipendenti dal particolare problema trattato. Di seguito si farà riferimento al caso di ordinamento di un array, sia per la maggiore semplicità, sia perché molti casi pratici sono ad esso riconducibili. Uno dei più semplici algoritmi di ordinamento è il seguente: si consideri il primo elemento, viene scorso il resto dell'array fino a trovare un elemento con valore minore, quindi si scambiano i due valori; si considera allora il successivo e così per ciascun elemento. Il codice che implementa questo algoritmo, per i dati considerati nel programma di esempio precedente, è riportato di seguito:

```
void ordina(Dato *tabella, int n)
{
  int i, j, min;
  Dato temp;

  for(i=0; i<n; i++)
  {
    min=i;
    for(j=i; j<n; j++)
      if (tabella[j].input<tabella[min].input) min=j;
    temp=tabella[i];
    tabella[i]=tabella[min];
    tabella[min]=temp;
  }
}
```

Il codice ordina i dati della tabella in base al valore crescente del campo `input`. Data la loro natura numerica l'operatore di confronto utilizzato è il semplice "minore" ("<"). Il primo `for` è utilizzato per determinare l'indice di partenza, il secondo per cercare il valore minimo nella parte restante dell'array. Va notato che l'indice del secondo for inizia dal valore attuale del primo. Questo significa che la parte "bassa" dell'array ad ogni passo risulta sempre ordinata. Un altro algoritmo di ordinamento è il cosiddetto "bubble sort", che esegue operazioni simili:

```
#define SWAP(tipo,a,b)   {tipo t; t=a; a=b; b=t;}

void bubble(Dato *tabella, int n)
{
  int i, j;

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

```
    {
      for(j=1;j<(n-i);j++)
      {
        if(tabella[j-1].input>tabella[j].input)
          SWAP(Dato, tabella[j-1].input,tabella[j].input);
      }
    }
}
```

L'algoritmo scorre l'array *n* volte scambiando tra loro i due elementi adiacenti se questi non risultano ordinati nel modo voluto.
Si può notare che in entrambi gli algoritmi vengono utilizzati due cicli `for` annidati, quindi per ordinare *n* elementi, è necessario eseguire approssimativamente n^2 operazioni! La complessità degli algoritmi di ordinamento semplici è infatti dell'ordine di n^2. Per dati di grandi dimensioni questo può essere un fattore molto limitante: per un array di 1000 elementi, potrebbero essere necessarie un milione di operazioni!
Si noti che un algoritmo come il *bubble sort* si presta bene ad essere applicato anche a liste, infatti sebbene non sia possibile accedere direttamente ai singoli elementi, è possibile conservare i puntatori che interessano (in particolare quello relativo all'elemento adiacente) quindi ricondursi ad una situazione simile a quella degli array.

L'ALGORITMO QUICKSORT

Per superare il limite della complessità quadratica degli algoritmi semplici di ordinamento, è possibile utilizzare un algoritmo noto come *Quicksort*, ideato da C. A. Hoare. Questo algoritmo è abbastanza complesso, ma ha il vantaggio fondamentale di riuscire a ordinare un insieme di dati con un numero di operazioni dell'ordine di $n\log_2 n$.
Il *Quicksort* utilizza una tecnica di tipo *divide et impera*, cioè scompone ricorsivamente l'intero insieme di dati in parti più piccole, che vengono ordinate separatamente in modo più semplice. L'algoritmo è disponibile come funzione nelle librerie standard dell'ANSI C e la funzione in questione di chiama `qsort` e risiede nella libreria *stdlib.h*. Il prototipo della funzione è il seguente:

```
void qsort(void *base,
           size_t nelem,
           size_t width,
           int (*fcmp)(const void *, const void *));
```

I parametri da passare alla funzione sono 4: il primo (`base`) è il puntatore al primo elemento dell'array da ordinare (ossia il nome dell'array stesso), il secondo (`nelem`) è un intero che definisce quanti sono gli elementi da ordinare, il terzo (`width`) indica la dimensione in byte di ciascun elemento dell'array ed il quarto, cioè `int (*fcmp)(const void *, const void *)` è a sua volta il prototipo di una funzione.
Nel caso della funzione `qsort` la funzione passata come parametro ha lo scopo di eseguire il confronto in base al quale si eseguirà poi l'ordinamento dei dati. Come già visto i dati da ordinare non hanno necessariamente una natura numerica o alfabetica quindi possono richiedere delle funzioni apposite per stabilire il significato di "precedere" o "seguire" (o simil-

mente “minore” o “maggiore”) all’interno di un insieme ordinato.
La funzione `qsort` richiede a sua volta una funzione che accetta i puntatori ai due elementi da confrontare e restituisce un valore intero che vale:

- < 0 se *elem1 “<” *elem2
- = 0 se *elem1 “=” *elem2
- > 0 se *elem1 “>” *elem2

dove il simbolo “<” significa “precede”, “>” significa “segue” e “=” invece “uguale/equivalente”. Quindi, per utilizzare la funzione `qsort` è necessario scrivere un’altra funzione che esegua il confronto tra due elementi e passarla come parametro.
Si consideri a titolo d’esempio il programma visto in precedenza ed il tipo di dati già utilizzato e si supponga di volere ordinare in modo crescente i dati contenuti nell’array (in base al valore del campo `.input`). Dal momento che il campo `.input` è di tipo numerico, per ottenere i valori voluti è sufficiente eseguire una sottrazione (ad esempio se `elem1<elem2` allora `elem1-elem2` sarà < 0 e così via). La funzione di confronto sarà la seguente (il nome della funzione è arbitrario):

```
int qcmp(Dato *a, Dato *b)
{
     return (a->input)-(b->input);
}
```

A questo punto sarà sufficiente richiamare la funzione `qsort` con i seguenti parametri:

```
qsort(tabella, N_ELEM, sizeof(Dato), qcmp);
```

Da notare che la dimensione degli elementi dell’array è stata ricavata utilizzando l’istruzione `sizeof`, mentre la funzione di confronto è stata passata semplicemente indicandone il nome. Questo è in accordo con quanto dichiarato nel prototipo della funzione `qsort`, perché in effetti i nomi delle funzioni in C, così come avviene nel caso degli array, sono dei puntatori e precisamente indicano l’indirizzo da cui inizia il codice ad esse associato.
Nel caso in cui sia necessario eseguire un confronto tra stringhe al fine di ordinarle alfabeticamente è possibile utilizzare la funzione `strcmp` della libreria *string.h*:

```
int strcmp(const char *s1, const char *s2);
```

che restituisce un parametro direttamente utilizzabile dalla funzione `qsort`. Per ottenere un ordine alfabetico inverso è sufficiente anteporre il segno meno al risultato restituito della funzione.

14. Aritmetica fixed point

INTRODUZIONE

In molte applicazioni può presentarsi la necessità di manipolare numeri non interi, cioè numeri "reali" o che comunque comprendono una parte frazionaria. Spesso sviluppando un programma in C l'approccio più semplice è quello di usare variabili di tipo *floating point* per trattare queste grandezze. Questa soluzione sebbene molto comoda (dal punto di vista di un programmatore ad alto livello), comporta una serie di implicazioni non indifferenti in pratica, che devono essere invece considerate per ottenere un'implementazione efficiente. Innanzi tutto i calcoli che coinvolgono numeri floating point richiedono un tempo di esecuzione sensibilmente più grande delle equivalenti operazioni svolte in aritmetica intera. Secondariamente la memorizzazione dei numeri floating point nella forma più semplice supportata dall'ANSI C (`float`) richiede 32 bit, una quantità di memoria che in molti casi è superiore a quella effettivamente richiesta dai calcoli che si stanno eseguendo. L'aumento dei tempi di elaborazione e della quantità di risorse di memoria richieste possono essere notevoli, perfino su macchine dotate di grandi risorse e capacità di calcolo, come i normali PC.
Ovviamente la situazione risulta ancora più grave quando il programma dovrà funzionare su piccoli sistemi embedded, tipicamente basati su microcontrollore. In questi casi infatti raramente si può contare sull'ausilio di un'unità a virgola mobile (FPU) quindi le operazioni sui tipi `float` saranno svolte da apposite librerie software, con un enorme incremento dei tempi di esecuzione. Un ulteriore svantaggio è rappresentato dal fatto che l'uso di aritmetica floating point comporta un maggiore consumo di potenza che si traduce in un maggiore assorbimento di corrente, con un conseguente aumento della dissipazione termica ed in generale una riduzione dell'autonomia di eventuali batterie. Come se non bastasse infine è abbastanza difficile convertire in ASCII un numero rappresentato in floating point, a meno di non usare librerie dedicate che solitamente sono piuttosto "voluminose".
Nella maggior parte dei casi è comunque possibile fare a meno dell'uso dell'aritmetica floating point, approssimandola con quella intera. I vantaggi offerti da questa tecnica possono essere enormi, basti pensare ad esempio alla possibilità di utilizzare funzioni trigonometriche su processori capaci di manipolare solo numeri interi. Questa tecnica verrà descritta nei successivi paragrafi ed è nota come aritmetica a "virgola fissa" (*fixed point* in inglese).

FIXED POINT IN BASE 10

Si supponga di dovere leggere il valore fornito da un sensore di temperatura e di doverlo visualizzare su un display in gradi Celsius nell'ipotesi che il range di temperatura in cui opera il sensore sia compreso tra 0°C e 60°C e che il sensore fornisca in questo intervallo un valore ad 8 bit (quindi compreso tra 0 e 255). Per ottenere una indicazione in gradi Celsius è sufficiente moltiplicare il valore fornito dal sensore per 0.235 (cioè 60/255). La moltiplicazione per un numero frazionario apparentemente richiederebbe l'uso della rappresentazione in virgola mobile, in realtà poiché la cifra visualizzata sarà sempre compresa tra "00.00" e "60.00" è possibile utilizzare un numero intero compreso tra 0 e 6000 per rappresentarla visualizzando il punto decimale tre le prime due cifre e le seconde ed aggiungendo eventualmente degli zeri in testa.
Con questo accorgimento i calcoli coinvolgono esclusivamente valori interi. Il risultato può essere ottenuto ad esempio utilizzando la seguente espressione:

```
T = (Dato * 235)/10
```

Ovviamente si tratterà di un valore approssimato, ma comunque accettabile per l'applicazione.
Applicando questa tecnica è sempre necessario nel corso delle operazioni tenere traccia "manualmente" della posizione del punto decimale e fare in modo che la sua posizione negli operandi sia correttamente allineata. Normalmente viene utilizzata la notazione S.X.Y, per indicare che il numero è dotato di segno (S), di X cifre a sinistra del punto decimale (intere) e di Y cifre a destra (frazionarie). Ad esempio per un numero composto da 5 cifre intere (in base 10), il suo formato sarà indicato come S.5.0, mentre un numero composto da 3 cifre intere e 2 decimali sarà indicato come S.3.2. Normalmente il numero di cifre complessivo utilizzabile è fissato, ad esempio un numero intero a 16 bit (uno `short int`) avrà al massimo 5 cifre (in base 10).
Queste potranno essere utilizzate per rappresentare sia la parte intera che quella frazionaria, fissando implicitamente la posizione del punto decimale. Alcuni esempi di "conversione" sono mostrati di seguito, considerando una rappresentazione S.3.2:

```
5.81    →  ‡ 581
-142.25 →  -14225
9.3     →  930
```

In questo caso per ottenente dei numeri interi è stata eseguita una moltiplicazione per 10 elevato al numero di cifre frazionarie. Questi numeri, essendo nello stesso formato possono essere sommati o sottratti con i normali operatori interi, supportati direttamente a qualsiasi processore.
Nel caso di numeri in formato diverso, prima di eseguire qualsiasi operazione occorre allineare la posizione dei loro punti decimali per ricondurli allo stesso formato. Si consideri ad esempio il numero 91.21 in formato S.3.2 e 54.1 in formato S.4.1 prima di eseguire la somma. Gli interi considerati sono:

```
91.21   →  9121 (S.3.2)
24.1    →  241 (S.4.1)
```

Per allineare correttamente i due valori è necessario moltiplicare per 10 quello in formato

S.4.1 e questo risulta chiaro in quanto il punto implicito del primo si trova prima delle cifre "21", mentre nel secondo si trova prima dell'"1" finale. I numeri interi da sommare sono quindi 9121 e 2410 (ora entrambi nel formato S.3.2), che forniscono un risultato pari a 11531, che va interpretato anch'esso come numero in formato S.3.2, cioè 115.31.
È possibile estendere i ragionamenti fatti anche alle altre operazioni aritmetiche. Il procedimento sarà comunque spiegato in maniera più completa nel prossimo paragrafo, con riferimento all'aritmetica fixed point binaria.

FIXED POINT IN BASE 2

Il fatto di dovere moltiplicare o dividere per potenze di 10 per allineare i numeri, rende l'aritmetica fixed point decimale poco efficiente, in quanto non tutti i processori supportano direttamente le operazioni di moltiplicazione e divisione. Questo limite non sussiste se si applicano le considerazioni precedenti a numeri in base due: le cifre da considerare sono i bit che compongono il numero e le potenze per cui moltiplicare e dividere saranno quelle di 2. Quest'ultimo particolare risulta molto importante, infatti moltiplicare o dividere un numero binario per potenze di due significa semplicemente scorrere i suoi bit a destra o a sinistra! Lo svantaggio in questo caso è che il numero intero ottenuto, visivamente non somiglierà affatto all'originale, quindi risulterà più difficile da visualizzare su un display, ma sarà comunque adatto ad eseguire i calcoli intermedi di un algoritmo e risulterà anche più preciso di quello decimale. Proprio per la maggiore efficienza, l'aritmetica fixed point binaria è in effetti la più utilizzata in pratica.
Si supponga di utilizzare numeri a 16 bit: la corrispondenza tra cifre e bit è diretta, quindi con il formato S.7.8 si indica un numero binario dotato di un bit di segno, 7 bit per la parte intera ed 8 per quella frazionaria. Ad esempio, i numeri 49.3 e 19.73 possono essere espressi in formato fixed point nel seguente modo:

```
19.73   →  19.73*2⁸ =    5051 (S.7.8)
-49.3   →  49.3 *2⁸ = -12621 (S.7.8)
```

Per ottenere questi numeri interi è stato sufficiente moltiplicare per 2 elevato al numero dei bit frazionari, cioè 8 in questo caso. I due numeri ottenuti possono essere sommati o sottratti, ottenendo ancora un numero intero, che deve essere interpretato nel formato S.7.8 (cioè va diviso per 2^8=256 per ottenere l'equivalente decimale):

```
   5051+ (S.7.8)
-12621= (S.7.8)
  -7570  (S.7.8)
```

Dividendo -7570 per 256 si ottiene -29.57, che è il risultato corretto della somma dei due numeri originali. In questi calcoli è stato necessario troncare o arrotondare i risultati parziali. Questo è dovuto al fatto che la parte frazionaria decimale necessita in genere di un numero di cifre binarie frazionarie maggiore per essere rappresentata senza errori. Disponendo di *n* cifre binarie per la parte frazionaria l'errore di rappresentazione sarà dell'ordine dell'LSB, cioè di $1/2^n$ (nel caso di 8 cifre sarà di circa 1/256 = 0.0039). Questo errore può essere ridotto di metà se prima di troncare il numero si somma il valore di LSB (cioè $1/2^{n+1}$). Come nel caso della rappresentazione fixed point decimale, anche in binario è necessario

fare in modo che i due numeri siano allineati prima di eseguire le operazioni di addizione o sottrazione. Si consideri il numero 5.41 in formato S.7.8 ed il numero 1.243 in formato S.5.10, allora sarà necessario moltiplicare il primo per 2 elevato a 2 (10-8=2), cioè 4, oppure dividere il secondo per lo stesso fattore. Il risultato sarà in formato S.5.10 nel primo caso o S.7.8 nel secondo. Di seguito sono mostrati i due casi:

```
5.41 * 256 = 1385     (S.7.8)
1.243 * 1024 = 1273   (S.5.10)
1385*4 + 1273 = 6813 (S.5.10)
6813 / 1024 = 6.653   (dec)
```

oppure

```
5.41 * 256 = 1385     (S.7.8)
1.243 * 1024 = 1273   (S.5.10)
1385 + 1273/4 = 1703 (S.7.8)
1703 / 256 = 6.652    (dec)
```

La scelta di uno dei due formati dipende da quale parte del numero sia più significativa dal punto di vista dell'applicazione (il primo ad esempio conserva una maggiore precisione sulle cifre frazionarie).

Moltiplicazione

La moltiplicazione è forse l'operazione più interessante, ma anche la più delicata da gestire. Il problema della moltiplicazione è che il numero di bit del risultato aumenta rispetto a quello degli operandi, quindi anche il formato risulta modificato. In particolare il prodotto tra due numeri fixed point di formato S.X.Y ed S.Z.W avrà il formato SS.(X+Z).(Y+W). Saranno dunque presenti due bit di segno e le parti intera e frazionaria saranno ampie quanto la somma delle ampiezze delle rispettive parti degli operandi. Questo ha due conseguenze principali: 1) i risultati intermedi devono essere contenuti in variabili dotate di un numero di bit maggiore di quelle di partenza (tipicamente larghe il doppio); 2) sono necessarie delle operazioni di troncamento/arrotondamento e divisione per riportare il risultato al formato originario.
Come esempio si consideri la coppia di numeri 5.41 e 1.243 visti sopra, espressi in formato S.7.8:

```
  1385x (S.7.8)
   318= (S.7.8)
440430  (SS.14.16)
```

Per riportare il risultato al formato di partenza (S.7.8) è necessario troncare la parte frazionaria ai primi 8 bit (scorrendo a destra di 8 bit, cioè dividendo per 256) e assicurarsi che la parte intera non necessiti di più di 7 bit per essere rappresentata (i bit in più verranno troncati dal momento che il risultato sarà collocato in una variabile a 16 bit):

```
440430/256 = 1720 (S.7.8)
1720/256 = 6.718  (dec)
```

Se i due operandi hanno formati diversi non ci sono particolari problemi, le dimensioni delle

due parti del risultato saranno sempre date dalla somma di quelle degli operandi, per cui sarà sufficiente scegliere un opportuno fattore di divisione per riportare il risultato al formato voluto. Ad esempio:

```
  (S.7.8)x
(S.5.10)=
(SS.12.18)/2^10=(S.7.8)
```

Divisione

Utilizzando l'aritmetica fixed point è possibile eseguire in due modi diversi le divisioni. Uno di questi è paragonabile al metodo tradizionale, l'altro invece risulta molto più vantaggioso, quando applicabile. Considerando la divisione normale, si nota che si verifica subito un effetto indesiderato: essa infatti ha l'effetto di cancellare i fattori costanti che ci hanno permesso di rendere interi i numeri su cui vogliamo operare! Cioè si ha che:

$$X*2^n/Y*2^n = X/Y$$

Se, come supporto all'inizio, possiamo effettuare soltanto divisioni intere, otterremo ogni volta la perdita di tutti i bit frazionari! Ad esempio:

```
1385/ (S.7.8)
 318= (S.7.8)
   4  (S.7.0)
```

Per evitare questo effetto occorre moltiplicare il dividendo per una potenza di due adatta a compensare la perdita dei bit, questa sarà pari al numero di bit frazionari che si vogliono ottenere:

```
1385*256 = 354560 (S.7.16)

354560/ (S.7.16)
   318= (S.7.8)
  1115  (S.7.8)
```

Il problema in questo caso è che per memorizzare il dividendo "allargato" occorre utilizzare variabili più grandi di quelle degli operandi (tipicamente il doppio), un po' come succede per la moltiplicazione.
Un altro problema caratteristico di questo approccio (che si ha anche lavorando con semplici numeri interi), è che la divisione, anche quando è supportata dall'hardware risulta piuttosto lenta rispetto alle altre operazioni (anche decine di volte!). Questo problema può essere superato utilizzando un metodo diverso per eseguirla. Infatti dividere un numero R per una costante S, equivale a moltiplicare R per 1/S. Ad esempio per sapere a quanti giorni corrispondono 105 ore, anziché dividere 105 per 24, lo si può moltiplicare per 1/24=0.041666 ed utilizzando una rappresentazione S.7.8 per entrambi i numeri si ottiene:

```
105*2^8 = 26880      (S.7.8)
0.041666*2^8 = 11    (S.7.8)
26880*11 = 295680   (SS.14.16)
```

```
295680/2^8 = 1155   (S.7.8)
1155/2^8 = 4.51     (dec)
```

Il risultato corretto sarebbe 4.375, quindi il valore ottenuto soffre un po' del ristretto numero di bit utilizzati nei diversi passi per rappresentare la parte frazionaria. Visti i numeri in gioco si sarebbe potuto utilizzare dei formati tagliati su misura per i rispettivi operandi, ad esempio S.15.0 per il primo, dal momento che è già un numero intero ed S.0.15 per il secondo, che invece è puramente frazionario. Il risultato è il seguente:

```
105                  (S.15.0)
0.041666*2^15 = 1365 (S.0.15)
105*1363 = 143325    (SS.15.15)
143325/27 = 1120     (S.7.8)
1120/256 = 4.375     (dec)
```

Questo esempio mette in luce che la scelta del formato da utilizzare in un determinato algoritmo deve tenere conto delle caratteristiche dei numeri trattati, della possibilità che si verifichino degli overflow, della disponibilità di memoria e del supporto da parte dell'hardware per l'esecuzione delle operazioni nei vari formati.
Va ricordato infine che è possibile applicare le tecniche viste anche a numeri privi di segno, l'unica differenza consiste nel fatto di disporre di un bit in più, che può essere sfruttato per estendere la larghezza dell'intervallo rappresentato (parte intera) o la sua precisione (parte frazionaria).

Q-FORMAT E ARITMETICA FRAZIONARIA

Eseguendo operazioni di moltiplicazione successive sugli stessi numeri, può capitare che la parte intera cresca a tal punto da non essere più rappresentabile col numero di bit di cui si dispone. Per evitare questo problema si può procedere in due modi: si dividono i risultati intermedi per 2 (eseguendo degli scorrimenti a destra); si utilizzano numeri privi della parte intera, quindi compresi nell'intervallo [-1,1]. La scelta di una delle due soluzioni dipende ovviamente dal particolare algoritmo da implementare. In questo paragrafo verrà analizzato il secondo metodo, spesso indicato col nome di "Q-format", che è di gran lunga il più utilizzato. Potere rappresentare delle grandezze nell'intervallo [-1,1] risulta fondamentale in moltissime applicazioni, infatti permette ad esempio di rappresentare segnali normalizzati di diverso tipo (audio, video…), ma anche funzioni trigonometriche come seno e coseno.
Per questo motivo il Q-format è supportato direttamente in hardware da quasi tutti i DSP e da alcuni microprocessori e microcontrollori.
Il Q-format consiste in una rappresentazione del tipo S.0.N, in cui tutti i bit, tranne quello di segno, sono utilizzati per rappresentare la parte frazionaria. Questo fa sì che in realtà l'intervallo rappresentabile sia compreso tra -1 e poco meno di +1. Uno dei formati più utilizzati è il già citato S.0.15 (detto anche Q-15).
La somma e la sottrazione tra numeri in Q-format è identica al caso intero, infatti sono utilizzabili i normali operatori interi e a differenza di quanto visto in precedenza tutti i numeri risultano sempre allineati, anche se esiste la possibilità di overflow se il risultato supera i limiti rappresentabili. La moltiplicazione invece, come già accennato, normalmente non genera overflow in quanto moltiplicando due numeri appartenenti all'intervallo [-1,1], il risultato

appartiene ancora a questo intervallo! Esiste solo un'eccezione che si verifica nella moltiplicazione -1 per -1. Il risultato dovrebbe essere 1, ma per quanto detto questo numero non è rappresentabile. Il più grande numero positivo (nel caso di Q-15) infatti è:

$(2^{15}-1)/2^{15} = 0{,}99996$.

La moltiplicazione tra due numeri Q-15 si avvale del seguente algoritmo: vengono moltiplicati i due numeri S.0.15 (16 bit) ottenendo un numero SS.0.30 (32 bit), come già visto. Il risultato è anch'esso costituito soltanto da bit frazionari (a parte i due bit di segno), questo significa che è possibile trascurare i 15 bit meno significativi, ottenendo lo stesso numero frazionario con una precisione meno spinta (15 bit invece che 30). Per fare questo e riportare il numero al formato originario a 16 bit è quindi sufficiente scorrere a destra di 15 bit il risultato e considerare solo i 16 bit meno significativi (oppure scorrere di un bit a sinistra, per cancellare il bit di segno in più e considerare i 16 più significativi). Esempio:

```
 0.69376x (dec)
-0.50584= (dec)
-0.35093  (dec)

 0.69376x215 =  22733 (S.0.15)
-0.50584x215 = -16575 (S.0.15)
     22733x
    -16575=
-376799475 _ (SS.0.30)
       /2^15=
    -11499= (S.0.15)
  -0.35092  (dec)
```

La divisione in Q-format crea qualche problema, infatti è abbastanza probabile che due numeri contenuti nell'intervallo [-1,1] divisi tra loro risultino in un numero che sta al di fuori di questo intervallo. Non solo, anche i reciproci stanno necessariamente fuori dall'intervallo! L'unica soluzione possibile, è quella di passare ad una rappresentazione S.X.Y, quindi operare come descritto prima e ricondursi al formato Q-n alla fine, se possibile.
In realtà comunque, questo è un problema mal posto, perché uno dei vantaggi di lavorare nell'intervallo [-1,1] è proprio quello di potere fare a meno della divisione!
Tutti i procedimenti visti fino a qui sono utilizzabili anche nel caso di rappresentazioni caratterizzate da un numero diverso di bit, ad esempio Q-3, Q-7, Q-31…

ESEMPI DI APPLICAZIONE DELL'ARITMETICA FIXED POINT IN C

Esempio 1: Luminosità dei pixel di un'immagine

Un'esigenza abbastanza comune nell'elaborazione di immagini è quella di ricavare l'informazione sulla luminosità di un pixel (cioè il suo equivalente in scala di grigi) a partire dall'informazione sui colori. Normalmente il colore di ciascun pixel è espresso tramite una terna di valori che codifica le sue componenti rosso, verde e blu (RGB). L'informazione sulla luminosità può essere ricavata eseguendo una somma pesata delle tre componenti, utilizzan-

do come pesi la sensibilità dell'occhio umano alle tre lunghezze d'onda. Analiticamente quanto detto viene espresso con la seguente relazione:

```
Y = 0.299*R + 0.587*G + 0.114*B
```

Si noti come il verde (G) dia un maggiore contributo alla luminosità rispetto al rosso (R) o al blu (B). Si noti anche che i coefficienti sono normalizzati, cioè la loro somma dà 1. Di solito ciascun componente di colore è espresso da un valore intero ad 8 bit senza segno (cioè compreso tra 0 e 255) quindi, utilizzando la notazione introdotta nei precedenti paragrafi, esso può essere pensato espresso nel formato fixed point 8.0. Dal momento che le operazioni da compiere coinvolgono soltanto numeri positivi, è corretto pensare di rappresentare i coefficienti nel formato 0.8, moltiplicandoli quindi per 2^8=256. L'equazione vista prima si può quindi riscrivere come segue:

```
Y = 76*R + 150*G + 29*B;
```

Il risultato dell'operazione sarà un numero a 16 bit (dal momento che risultano a 16 bit i prodotti parziali), nel formato 8.8. Queste considerazioni indicano che è opportuno dichiarare Y come `unsigned short`. Per riportare il valore ottenuto al formato originario (8.0) è necessario considerare soltanto gli 8 bit superiori (parte intera), e questo può essere ottenuto dividendo per 256 o meglio eseguendo uno shift a destra di 8 bit e considerando soltanto gli 8 bit inferiori ottenuti:

```
Y = (unsigned char) Y>>8;
```

Il casting ad `unsigned char` non è strettamente necessario, perché se Y è stata dichiarata come `unsigned short`, lo shift a destra viene inteso come logico e non aritmetico quindi da sinistra sono introdotti degli zeri. Utilizzare una variabile a 16 bit per contenere la somma di tre prodotti anch'essi a 16 bit, in questo caso non genera un overflow perché i coefficienti utilizzati sono normalizzati, e questo fa sì che al massimo la somma dei tre prodotti può raggiungere il limite superiore rappresentabile.
Per provare l'algoritmo appena descritto, di seguito è riportato il codice di un programma che legge i dati dell'immagine da un file "Windows Bitmap" (.BMP) e, dopo avere ricavato i valori della luminosità, salva una seconda immagine convertita in scala di grigio.

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

// Lunghezza header
#define HLEN 57

// Definizione tipo di dati
typedef struct {
   unsigned char B;
   unsigned char G;
   unsigned char R;
} tipo_dati;

// Prototipi
```

```
tipo_dati GrayScale(tipo_dati);

// *** Main ***
void main()
{
  FILE *inputf, *outputf;
  tipo_dati dato_in, dato_out;
  int i;

  // Apre file
  inputf=fopen("Image.bmp", "rb");
  outputf=fopen("Image_mod.bmp", "wb");

  // Controllo errori
  if ((inputf==NULL)||(outputf==NULL)) {
     printf("Errore sui file!\n");
     exit(1);
  }

  printf("Caricamento...\n");

  // Copia header del file
  for(i=0; i< HLEN; i++)
    fputc(fgetc(inputf), outputf);

  // Loop principale
  while(!feof(inputf))
  {
     fread(&dato_in, sizeof(tipo_dati), 1, inputf);
     dato_out=GrayScale(dato_in);
     fwrite(&dato_out, sizeof(tipo_dati), 1, outputf);
  }

  // Chiusura file
  fclose(inputf);
  fclose(outputf);

  printf("Fine!\n");
}

tipo_dati GrayScale(tipo_dati col)
{
   unsigned short y;
   tipo_dati gr;

   y = 76*col.R+150*col.G+29*col.B;
   y >>= 8;

   gr.R = y;
```

```
    gr.G = y;
    gr.B = y;

    return gr;
}
```

Il programma utilizza due file come sorgente e destinazione per i dati, ma è stato scritto considerando un'elaborazione a dato singolo, come avviene di solito quando i dati provengono e sono diretti a delle porte o periferiche hardware. Inizialmente vengono aperti i file di origine e di destinazione dei dati, successivamente viene verificato che l'operazione sia andata a buon fine. Il file bitmap viene utilizzato come una sorgente di dati grezzi, quindi il suo header non viene decodificato (per leggere le caratteristiche dell'immagine), ma viene semplicemente copiato nel fine di destinazione. Per questo motivo il programma gestisce correttamente solo file a 24 bit non compressi. La lunghezza dell'header è definita dalla macro HLEN. In un file BMP a 24bit i dati dell'immagine sono organizzati in triple RGB, che iniziano subito dopo l'header e si concludono al termine del file. L'ordine in cui sono disposti i pixel non è rilevante dal momento che l'operazione avviene soltanto sui colori.
La funzione GrayScale è quella che esegue il calcolo della luminosità. Si ricorda che l'espressione y >>= 8, che implementa lo shifting di 8 bit a destra (ossia la divisione per 256) equivale alla più esplicita y = y >> 8.

Esempio 2: Filtraggio digitale

Una delle applicazioni più diffuse nel campo del *Digital Signal Processing* è quella legata alla realizzazione di filtri digitali. La trattazione dell'argomento è ben al di là degli scopi di questo libro, ma è sufficiente sapere che applicando formule simili a quella vista precedentemente è possibile realizzare in digitale gli stessi filtri che si realizzano comunemente tramite circuiti analogici. Di seguito verrà analizzata la realizzazione fixed point di un semplice filtro FIR (*Finite Impulse Response*) del 4° ordine, la cui struttura è mostrata in figura 14.1. Il funzionamento è il seguente: i campioni X vengono fatti scorrere attraverso una serie di stadi di ritardo, da cui sono prelevati i valori, che vengono moltiplicati per opportuni coefficienti quindi sommati assieme per ottenere l'uscita del filtro. Questo procedimento può essere descritto sinteticamente dalla seguente espressione:

```
Y = X[t]*W[0]+ X[t-1]*W[1]+ … + X[t-4]*W[4]
```

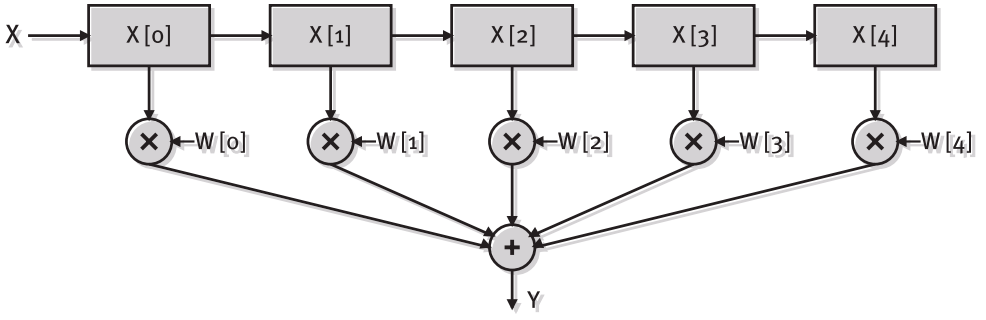


Figura 14.1
Struttura del filtro FIR

Nella formula i termini X[i] rappresentano i campioni del segnale da filtrare al tempo k(t-i), mentre i termini W[i] sono i coefficienti che moltiplicano i diversi campioni ritardati. La scelta dei coefficienti W[i] dipende dal tipo di filtro da realizzare. In questo esempio è stato scelto di realizzare un filtro passa basso con frequenza di taglio a circa il 25% della banda del segnale (che sarà compresa tra 0 e la metà della frequenza di campionamento, per il teorema di Nyquist). Il valore dei coefficienti è stato determinato utilizzando il noto programma Matlab, che come è visibile in figura 14.2 ha permesso anche di verificare quale sarà l'impatto dell'utilizzo dell'aritmetica fixed point (cioè della quantizzazione dei coefficienti). I coefficienti trovati sono i seguenti:

```
W[0] = 0.0781250
W[1] = 0.2265625
W[2] = 0.3828125
W[3] = 0.2265625
W[4] = 0.0781250
```

Per testare il programma i dati da elaborare verranno ricavati da un file audio in formato .WAV con campioni ad 8 bit. La frequenza di campionamento non ha importanza in quanto il filtraggio sarà sempre fino al 25% della banda. Per questo tipo di applicazione di solito viene scelto il Q-format, quindi i segnali ed i coefficienti sono considerati come numeri dotati soltanto della parte frazionaria quindi appartenenti all'intervallo [-1,1]. I campioni ad 8 bit possono essere considerati già espressi in questo formato (Q-7), infatti equivalgono a numeri compresi in [-1,1] moltiplicati per 2^7=128. I coefficienti invece devono essere moltiplicati per 128, ottenendo i seguenti valori:

```
W[0] = 10
W[1] = 29
W[2] = 49
W[3] = 29
W[4] = 10
```

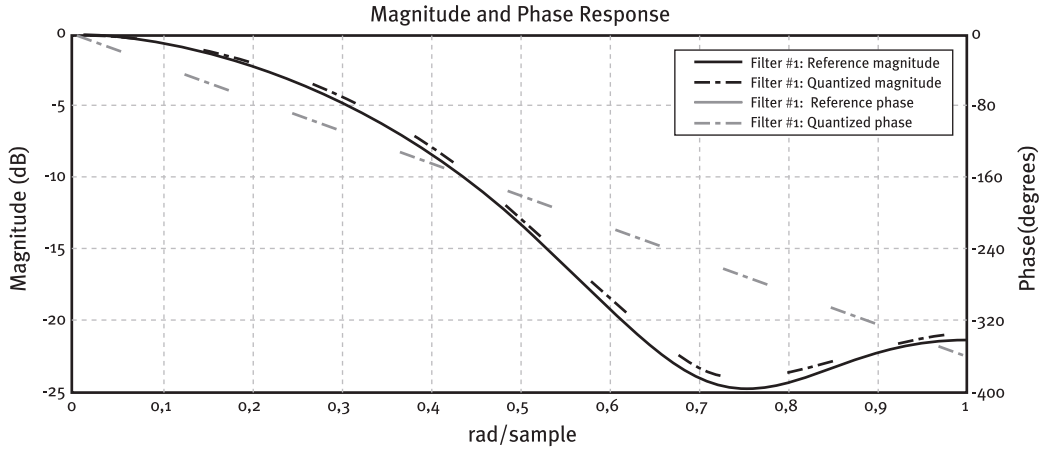


Figura 14.2
Funzione di trasferimento del filtro calcolata con Matlab

La funzione che implementa il filtro è la seguente:

```
tipo_dati FIR(tipo_dati samp)
{
  static char taps[5];
  char i, w[5]={10, 29, 49, 29, 10};
  short int y;

  // Scorre i campioni nei ritardi
  taps[0]=samp-128;
  for(i=4; i>0; i--)
    taps[i]=taps[i-1];

  y=taps[0]*w[0]+
    taps[1]*w[1]+
    taps[2]*w[2]+
    taps[3]*w[3]+
    taps[4]*w[4];

  return ((unsigned char) (y>>7))+128;
}
```

Questa funzione può essere inserita nel programma visto precedentemente sostituendo il nome della funzione richiamata dal `main`, ridefinendo la lunghezza dell'header (HLEN) a 44 ed il tipo di dati come segue:

```
typedef unsigned char tipo_dati;
```

Anche in questo caso l'header non viene decodificato, ma solo copiato, quindi possono essere utilizzati soltanto campioni PCM ad 8 bit.
La funzione è stata scritta in modo da potere essere richiamata per ogni campione ottenuto e si presta quindi ad essere utilizzata anche su un microcontrollore, in cui i campioni possono essere letti direttamente dall'ADC.
I file WAV utilizzano campioni ad 8 bit `unsigned`, che per rispettare il formato previsto da programma devono quindi essere resi `signed`, sottraendo 128 quindi essere di nuovo convertiti in `unsigned` alla fine dei calcoli. I campioni vengono prima di tutto fatti scorrere negli stadi di ritardo del filtro (implementati con un semplice array), poi vengono utilizzati per calcolare l'uscita. L'array che li memorizza è stato dichiarato come `static` in modo da non essere perso quando si esce dalla funzione. L'uscita del filtro è calcolata usando la formula vista prima. La moltiplicazione tra i due numeri Q-7 (quindi S.0.7), come già detto fornisce un numero a 16 bit in formato Q-14 (SS.0.14). Per riportare il risultato al formato originario è necessario scorrere di 7 bit a destra. Anche in questo caso il risultato complessivo è memorizzato in uno `short int` e non si possono verificare overflow perché i coefficienti sono normalizzati. Si può notare inoltre che per la simmetria del filtro i coefficienti 0 e 4 ed 1 e 3 sono uguali: si sarebbe potuto sfruttare questa simmetria per eseguire due moltiplicazioni in meno! (Sommando prima i rispettivi due campioni).

Esempio 3: Rotazione di vettori

La trasformazione lineare che esprime la rotazione di un vettore è la seguente:

```
xr = x*cos(a) – y*sin(a)
yr = x*sin(a) + y*cos(a)
```

Dove `x` ed `y` sono le coordinate iniziali del punto, mentre `xr` ed `yr` quelle ottenute in seguito alla rotazione di un angolo `a`. Per il calcolo dei valori del seno e del coseno è conveniente utilizzare una `look-up table` (LUT), un array che contiene i valori precalcolati di seno o coseno relativi all'intervallo di angoli che interessano. Sfruttando la simmetria delle funzioni seno e coseno e la loro similitudine (a meno si uno sfasamento di 90°) è possibile ridurre sensibilmente il numero di valori nella tabella, che sarà comunque determinato anche dalle esigenze di precisione richieste. I valori di seno e coseno sono compresi tra –1 ed 1, è quindi possibile renderli in fixed point con una rappresentazione del tipo S.0.X. Un esempio di tabella contenente 7 voci relative al seno di angoli compresi tra 0 e 90° può essere la seguente:

```
0°  →  0.00000000
15° →  0.25881904
30° →  0.50000000
45° →  0.70710678
60° →  0.86602540
75° →  0.96592582
90° →  1.00000000
```

che espressa in fixed point, utilizzando una rappresentazione S.0.7 diviene:

```
0°  →  0
15° →  33
30° →  64
45° →  90
60° →  111
75° →  124
90° →  127
```

Un esempio di funzione per il calcolo della rotazione di un vettore è la seguente:

```
void Rotate(short int *x, short int *y, unsigned char a)
{
  short int xt, yt;
  char sint[7]={0, 33, 64, 90, 111, 124, 127};

  xt = ((*x) * sint[6-a] – (*y) * sint[a])>>7;
  yt = ((*x) * sint[a] + (*y) * sint[6-a])>>7;

  *x=xt;
  *y=yt;
}
```

Le coordinate del punto da ruotare sono passate alla funzione per riferimento, in questo modo il valore calcolato può essere restituito nelle stesse variabili. L'angolo di rotazione è dato direttamente come indice dell'array che riporta il valore del seno degli angoli.

Ovviamente esistono delle soluzioni molto più eleganti, come ad esempio calcolare un'approssimazione dell'angolo o addirittura eseguire un'interpolazione lineare tra i valori tabellati. Il valore del coseno è stato calcolato a partire da quello del seno, utilizzando la relazione che intercorre tra i due. Va notato che la larghezza in bit e la rappresentazione delle coordinate del vettore non è rilevante al fine dei calcoli, infatti il modulo del vettore non è influenzato dalla rotazione e quindi il suo formato rimane praticamente inalterato. Nel codice ad esempio il vettore ha coordinate espresse nel formato S.15.0, mentre seno e coseno sono espressi come S.0.7. Effettuato lo shifting dopo le moltiplicazione, si riconduce il numero ottenuto dal formato SS.15.7 a quello originario S.15.0. L'errore commesso rispetto all'uso della rappresentazione floating point dipende ovviamente dal numero di bit che si utilizzano per rappresentare i coefficienti e dal modulo del vettore. In figura 14.3 è visibile il risultato della rotazione di un vettore di modulo 150 effettuato utilizzando la rappresentazione flotaing point (linea continua) e quella fixed point ad 8 bit utilizzata nell'esempio (linea tratteggiata). Si noti come l'errore sia relativamente contenuto. Utilizzando una rappresentazione a 16 bit l'errore risulterebbe quasi invisibile.
Queste tecniche, sebbene apparentemente molto semplici, hanno diverse applicazioni importanti: dalla generazione di segnali sinusoidali, alle modulazioni e demodulazioni digitali, al calcolo della trasformata di Fourier di un segnale e molte altre.

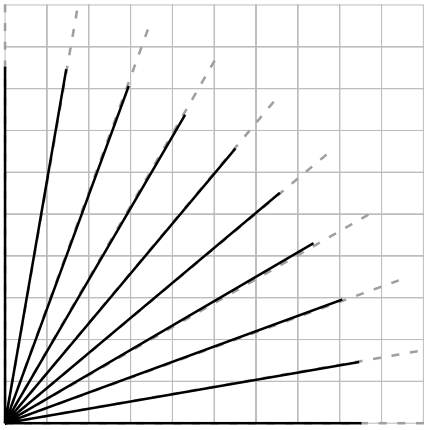


Figura 14.3
Rotazione di un vettore in floating point (linea continua) e fixed point (linea tratteggiata)

15. Ottimizzazione del codice

INTRODUZIONE

Un codice ottimizzato può essere eseguito efficientemente anche su un processore poco potente, quindi meno costoso, in altri casi invece può permettere di soddisfare vincoli temporali stringenti. L'ottimizzazione del codice non è una disciplina precisa e ben codificata, è piuttosto una specie di "arte" che si impara soltanto con molta pratica ed esperienza. Quando si applicano queste tecniche però devono essere tenuti in considerazione anche gli eventuali effetti indesiderati che esse introducono. L'uso di alcune di queste tecniche spesso comporta la violazione di molte delle regole che di solito sono considerate di "buona programmazione" e questo potrebbe avere effetti negativi sia sulla leggibilità del codice, sia durante la fase di debug (è più difficile trovare e correggere gli errori). Alcune tecniche riescono ad incrementare la velocità di esecuzione a discapito della compattezza del codice, occorre pertanto valutare bene la loro applicazione, soprattutto quando la disponibilità di memoria per memorizzare il codice non è adeguata o comunque l'uso di memoria aggiuntiva risulta costoso.
In generale l'incremento della velocità di esecuzione viene ottenuto minimizzando il numero di istruzioni macchina da eseguire per una determinata routine o diminuendo il più possibile il numero delle operazioni "lente". Per fare questo è dunque necessario conoscere come il compilatore traduce il nostro codice C in codice macchina.

IL COMPILATORE

La maggior parte dei compilatori offrono già una serie di opzioni per incrementare le prestazioni a run-time del codice. Un primo passo per ottenere codice macchina più ottimizzato consiste quindi nell'abilitare queste opzioni. In questo caso non c'è un intervento sul codice C, ma viene forzato il compilatore ad eseguire un lavoro più accurato e minuzioso nella fase di traduzione dal codice C in codice macchina. Il risultato di questa ottimizzazione è spesso cumulabile con quello che si può ottenere agendo manualmente sul codice. In genere è possibile selezionare un'ottimizzazione mirata alla compattezza del codice generato, oppure alla velocità di esecuzione.

Di solito non è possibile ottenere automaticamente entrambe, oppure è possibile solo a spese di un aumento dei tempi di compilazione. Le opzioni disponibili e la qualità dei risultati dipendono molto sia dal compilatore utilizzato, sia dalla particolare architettura del processore target. In figura 15.1 è visibile la finestra del compilatore Dev-C++ relativa proprio alle ottimizzazioni: sono disponibili tre livelli, di complessità ed efficacia via via crescente. Le stesse opzioni si possono trovare nel noto compilatore GCC (disponibile per diversi processori e microcontrollori), utilizzando i parametri -O, -O2 e -O3.

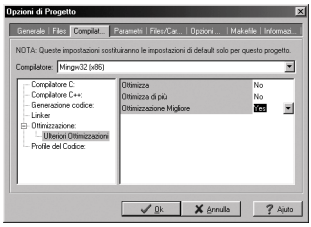


Figura 15.1
Finestra relativa alle ottimizzazioni del compilatore Dev-C++

In genere le ottimizzazioni più comuni mirano ad utilizzare il minor numero di istruzioni di salto possibile, il minor numero di accessi in memoria o a raggruppare costanti o sottoespressioni. I primi due accorgimenti sono dovuti al fatto che le istruzioni di salto e di accesso alla memoria richiedono spesso più cicli di clock rispetto alle altre, per cui risulta vantaggioso evitarle o aggirarle. Questo si può ottenere ad esempio utilizzando maggiormente i registri rispetto alla memoria per manipolare le variabili, oppure ripetere esplicitamente delle istruzioni invece di eseguire dei loop e quindi dei salti (*loop unrolling*). Il raggruppamento delle espressioni comuni invece permette sia di ridurre il numero di accessi in memoria, sia di eseguire una sola volta i calcoli quando si trovano espressioni identiche ripetute.
Quando vengono abilitate diverse ottimizzazioni automatiche, è difficile prevedere il risultato della loro interazione e, anche se il codice prodotto in genere funzionerà in modo identico a quello non ottimizzato (a parte la differenza in velocità), può capitare che il compilatore faccia delle assunzioni sul codice sorgente, che non erano nelle intenzioni del programmatore! Un esempio classico e abbastanza illuminante a tal proposito è il seguente. Si supponga che all'interno di un programma eseguito su un microcontrollore ad un certo punto occorra attendere la pressione di un tasto per continuare. Per fare questo si può utilizzare un *polling*, leggendo all'interno di un ciclo `while` la locazione di memoria che corrisponde alla porta di I/O a cui fa riferimento il pulsante.
Il codice è il seguente:

```
#define KEY *((unsigned char *) 0xFF00)
…
// Attesa pressione tasto
while(!KEY) {}
…
```

La macro `KEY` corrisponde al contenuto della locazione di memoria 0xFF00, cioè quella in cui è mappata la porta di I/O. Si ipotizzi che se nessun tasto è premuto tutti i bit valgano 0. L'istruzione `while` utilizzata per implementare l'attesa controlla il valore di `KEY` e fino a quando il valore di questa resta 0, ripete il codice tra parentesi. Quando verrà premuto un tasto la condizione non sarà più verificata quindi terminerà il loop. Per eseguire corretta-

mente il controllo la locazione di memoria dovrebbe essere letta in continuazione, in quanto essa potrebbe cambiare in un qualsiasi momento alla pressione di un tasto. Abilitando le ottimizzazioni, il compilatore deduce dal codice che, dal momento che nessuna istruzione cambia il valore della locazione di memoria, essa rimane inalterata, quindi inizialmente essa verrà copiata in un registro ed i successivi confronti verranno fatti con il valore memorizzato nel registro. Ovviamente anche dopo la pressione di un tasto il valore del registro rimarrà sempre lo stesso ed il programma non uscirà mai dal ciclo! Per evitare questa situazione occorre specificare che la locazione di memoria coinvolta può cambiare indipendentemente dal programma. Per fare questo si usa la keyword `volatile`.

```
#define KEY *((volatile unsigned char *) 0xFF00)
```

Diverse ottimizzazioni "semplici" (comunque riconducibili a quelle descritte prima) sono messe in atto normalmente dal compilatore, quindi non c'è bisogno di preoccuparsene in prima persona, anzi, si può sfruttare questo fatto per aumentare la leggibilità del codice. Ad esempio usando delle espressioni aritmetiche che hanno un risultato costante, queste non vengono valutate a run-time (riducendo l'efficienza), ma il risultato viene calcolato già durante la fase di compilazione e sostituito all'espressione. Quindi scrivere:

```
for(i=0; i<8*8*2; i++)
…
```

non risulta meno efficiente di

```
for(i=0; i<128; i++)
…
```

IL CODICE

In questo paragrafo verrà illustrato come intervenire sul codice per aumentarne l'efficienza.

Chiamate a funzioni

La chiamata ad una funzione nel codice C viene di solito tradotta in linguaggio macchina utilizzando delle istruzioni di salto e delle istruzioni che accedono alla memoria per eseguire delle operazioni sullo stack per gestire gli argomenti passati o restituiti. Questi due tipi di istruzioni, come già detto, risultano di solito più lente delle altre e quindi il loro uso dovrebbe essere limitato. Una prima soluzione è quella di utilizzare macro al posto delle funzioni, quando possibile. Questo fa in modo che invece di richiamare una funzione, le istruzioni vengano copiate per esteso quando e dove servono. Ovviamente questo comporta un incremento della lunghezza del codice, quindi è consigliabile usare questo accorgimento per funzioni semplici o che sono richiamate in pochi punti del codice. Ad esempio la funzione che esegue il test di un bit di una variabile:

```
int BitTest(int v, char pos)
{
  return (v>>pos)&1;
}
```

può essere vantaggiosamente definita come macro:

```
#define BitTest(v, pos)  (v>>pos)&1
```

Un altro metodo, utilizzabile però solo con alcuni compilatori (non è una keyword standard) per ottenere lo stesso risultato è quello di inserire l'istruzione `inline` prima della funzione:

```
inline int BitTest(int v, char pos)
```

Uso dei registri

Per limitare il più possibile gli accessi in memoria, le variabili utilizzate più frequentemente dovrebbero essere copiate nei registri del processore prima di eseguire le operazioni che le coinvolgono. Per fare questo le variabili devono essere segnalate al compilatore ed in secondo luogo le operazioni da svolgere devono essere in una forma tale da utilizzare il minor numero possibile di operandi ausiliari (questo perché il numero di registri disponibili è sempre molto limitato). Per indicare che una variabile è usata di frequente (quindi andrebbe copiata e tenuta nei registri) si aggiunge la keyword `register` alla sua dichiarazione:

```
register int x;
```

Dal momento che il numero di registri è di solito limitato l'uso della keyword `register` è considerato dal compilatore come un "suggerimento" più che un comando: sarà il compilatore stesso a scegliere se è il caso di accettarlo o meno. In ogni caso occorre tenere presente che un uso eccessivo della keyword `register` può anche avere effetti negativi: se vengono segnalate troppe variabili come `register`, il compilatore avrà difficoltà a capire quali sono quelle per cui convenga realmente effettuare l'operazione. Un numero eccessivo di variabili copiate nei registri potrebbero non lasciare posto a variabili molto più critiche, ma che non appaiono tali al programmatore. Infine è bene ricordare che in alcuni casi (ad esempio in certi microcontrollori) non c'è differenza tra i tempi di accesso dei registri e della RAM interna, quindi l'uso della keyword `register` potrebbe risultare del tutto inutile.

Uso delle variabili globali

Il passaggio dei parametri alle funzioni è di solito realizzato tramite lo stack, cioè le variabili sono memorizzate nello stack prima di chiamare la funzione. L'uso dello stack comporta un aumento degli accessi in memoria e quindi può rallentare le operazioni. Una soluzione semplice a questo inconveniente è quella di usare delle variabili globali, anziché passarle come argomenti. Si ricorda che le variabili globali sono definite fuori da qualsiasi funzione (compreso il `main`) e per questo sono visibili da qualsiasi punto del programma. Può essere vantaggioso quindi creare un set di variabili globali quando queste siano utilizzate come parametro da molte funzioni o si riferiscano a dati effettivamente condivisi dalle funzioni. L'uso delle variabili globali di solito è sconsigliato, perché può creare dei problemi di coerenza e può diminuire la leggibilità del codice. Inoltre anche il debug del programma risulta più difficoltoso: se si verifica un errore sui dati più essere più difficile capire chi o cosa lo ha generato (è anche vero comunque che proprio la loro visibilità le rende più osservabili).

Uso del goto

Un'altra pratica che di solito è sconsigliata è quella di utilizzare l'istruzione `goto`. Questa istruzione può creare gli stessi problemi visti prima a proposito delle variabili globali, ma risulta molto utile per velocizzare e semplificare il codice nei casi in cui sono coinvolti com-

plicati controlli o sequenze di codici simili ripetute. Anche se i salti sono istruzioni piuttosto lente, le semplificazioni introdotte dal `goto` possono dare un vantaggio tale da rendere conveniente il suo utilizzo. Ad esempio si consideri il seguente codice:

```
k=0;
for(i=0; i<100; i++) {
  for(j=24; j<200; j++) {
    if (w[j]==0) goto esci;
    k=k+v[i]/w[j];
  }
}
esci:
```

Sono stati utilizzati due cicli `for` annidati per eseguire un calcolo iterativo che prevede una divisione. Se il denominatore viene trovato uguale a 0 viene interrotto il calcolo e per uscire direttamente dai due cicli `for` è stato utilizzato un `goto`. Per evitare il `goto` sarebbe stato necessario eseguire due controlli, uno in ciascuno dei due loop, che sarebbero stati eseguiti ad ogni iterazione.

Dati, tipi e strutture

Per rendere il più possibile veloce l'esecuzione di operazioni aritmetiche, logiche o di movimento è consigliabile utilizzare il più possibile dei tipi di dati della lunghezza ottimale per la macchina su cui verrà eseguito il codice. Ad esempio molti microcontrollori ad 8 bit possono operare direttamente soltanto su dati ad 8 bit (ad esempio alcuni PIC), altri possono operare su dati sia da 8 che da 16 (es. Z80, AVR, 8051/2), mentre un microprocessore a 32 bit può quasi sempre gestire direttamente dati da 8, 16 e 32 bit, anche se probabilmente esegue le moltiplicazioni in un ciclo di clock solo su 8 bit (es. ARM7, MIPS). Utilizzare dati di larghezza maggiore di quella richiesta può comportare un certo rallentamento nelle operazioni di lettura e scrittura, inoltre può non permettere l'uso dei registri per memorizzare le variabili. In ogni caso, indipendentemente dalla larghezza dei dati, è comunque importante cercare di utilizzare i tipi nativi offerti dall'ANSI C, in caso contrario si è costretti ad eseguire del codice aggiuntivo per ogni operazione svolta sui dati, aumentando il tempo di esecuzione dell'operazione ed il numero di accessi in memoria. Una delle ragioni dell'inefficienza che si può originare nella gestione dei tipi non standard in C, è dovuto al fatto che risulta complicato reperire delle informazioni che in assembler sarebbero a disposizione automaticamente. Ad esempio, per implementare un'addizione a 64 bit, occorrerebbe conoscere il valore del flag di riporto del processore, che non è accessibile direttamente dal C; occorre quindi calcolarlo esplicitamente, aumentano il numero di istruzioni da svolgere. Un discorso a parte meritano i tipi strutturati. Essi sono implementati in maniera differente da ogni compilatore, anche in relazione all'architettura hardware. Questo implica che non sempre la selezione di un campo si traduce in una sola lettura dalla memoria, ma può comportare anche operazioni aggiuntive come shifting multipli e mascheratura. Questo si verifica in particolare quando la dimensione dei campi non coincide con la granulosità d'indirizzamento del microprocessore utilizzato. Si consideri ad esempio l'utilizzo di campi di bit:

```
typedef struct {
  char ID : 4;
  char Tipo : 5;
  short Lunghezza : 12;
} header;
```

Per ciascuno dei campi della struttura è stata definita la lunghezza in bit. Se il compilatore memorizza i campi in sequenza, per leggere il campo `ID` sarà necessaria una lettura ed una mascheratura (un AND con 0x0F), per leggere il campo `Tipo` invece, potrebbe essere necessaria (se i bit sono memorizzati in maniera contigua) la lettura di ben due byte, lo shifting di 4 posizioni a destra e la mascheratura. Alcuni compilatori in realtà non memorizzano sequenzialmente i bit, ma pongono i vari campi in locazioni indirizzabili singolarmente, lasciando quindi un po' di spazio "vuoto" tra loro. In questo caso non sarà necessario lo shifting, ma soltanto la mascheratura. Una situazione analoga si può verificare anche utilizzando i normali tipi, quindi è sempre il caso di valutare anche questo aspetto prima di utilizzare un certo tipo di dati.

Operazioni aritmetiche

Normalmente le operazioni aritmetiche non richiedono tutte lo stesso tempo di esecuzione, in genere l'addizione e la sottrazione possono essere eseguite in un solo ciclo di clock, la moltiplicazione può richiedere più di un ciclo, mentre la divisione (e anche il modulo %), perfino quando supportata dall'hardware, ne richiede sempre un numero abbastanza grande (decine). Da questo si intuisce che quando possibile la divisione deve essere evitata e l'uso della moltiplicazione deve essere ridotto.
In molti casi questo non è difficile e si può ottenere utilizzando qualche piccola accortezza, come mostrato nel seguente esempio, che mostra come scrivere un valore in una matrice bidimensionale di dimensioni 30x20, mappata in un'area di memoria lineare. In questo caso la difficoltà nasce dal fatto che le prime 10 colonne devono essere lasciate inalterate, quindi non è possibile soltanto incrementare il puntatore, ma è necessario utilizzare un "indirizzamento" riga-colonna:

```
  for(j=0; j<20; j++) {
    for(i=10; i<30; i++)
      a[i+30*j]=79;
}
```

Per ogni iterazione viene utilizzata una somma ed una moltiplicazione, per un totale di 20x30 operazioni solo per calcolare il valore dell'indice! L'uso della moltiplicazione può essere evitato riorganizzando il ciclo:

```
k=0;
for(j=0; j<20; j++) {
  for(i=10; i<30; i++)
    a[i+k]=79;
  k+=30;
}
```

Nel caso in cui le moltiplicazioni e divisioni coinvolgano potenze di due, è bene utilizzare le operazioni di bit shifting (<< e >>): uno scorrimento a destra equivale ad una divisione per 2, a sinistra ad una moltiplicazione per 2. Scorrendo più bit si possono ottenere le altre potenze di 2:

```
a = b*32;
```

equivale a:

```
a = b<<5;
```

Gli scorrimenti di una posizione sono eseguiti normalmente in un solo ciclo di clock, quindi più velocemente delle moltiplicazioni. Quelli di più posizioni possono richiedere tanti cicli quanti sono i posti da scorrere (a meno che il processore non disponga di un *barrel shifter*, in questo caso è necessario un solo ciclo di clock). Se il processore non è dotato di un moltiplicatore hardware allora risulta sempre conveniente usare gli scorrimenti, se ne è dotato allora occorre valutare bene quale dei due metodi risulti più vantaggioso.
Per quanto riguarda la divisione o altre operazioni intrinsecamente lente (operazioni trigonometriche, numeri casuali, funzioni complesse, etc.) è possibile spesso utilizzare delle *look-up tables*, ovvero tabelle che contengono i risultati precalcolati. In questo modo sarà possibile ottenere un risultato semplicemente leggendolo dalla tabella. Ovviamente questo metodo incrementa la dimensione del codice oggetto, ma può accelerare notevolmente l'esecuzione del programma ed è particolarmente adatto in quei casi in cui il numero di elementi della tabella ed il loro contenuto è noto con precisione a priori (es. calcolo della FFT). Inoltre, come già visto, per ottenere prestazioni accettabili nella maggior parte dei casi è necessario rinunciare all'uso dell'aritmetica floating point e sostituirla con quella fixed point (capitolo 14).

Librerie standard

Non sempre le librerie standard, messe a disposizione dai compilatori, implementano nel modo più efficiente le funzioni a cui sono preposte. Questo è dovuto, oltre che alle differenze d'implementazione da un compilatore all'altro, anche al fatto che le funzioni messe a disposizione cercano di gestire i casi più generali. Questo comporta non solo una potenziale inefficienza (o meglio, non ottimizzazione), ma anche un incremento di dimensioni del codice oggetto, spesso non necessario. In molti casi quindi è consigliabile riscrivere alcune funzioni in maniera molto più sintetica ed ottimizzata per la particolare applicazione. Si consideri ad esempio la funzione `memcpy` (libreria *strings.h*), che serve per copiare un blocco di dati da un'area di memoria ad un'altra. Essa in genere legge e scrive la memoria a byte. Se il processore è dotato di un indirizzamento a 32 bit, è possibile copiare la stessa quantità di memoria 4 volte più velocemente (copiando word da 32 bit invece che singoli byte). Qualora nel sistema sia presente un DMA, è molto più conveniente riscrivere una versione della `memcpy` che ne faccia uso ricorrendo all'*inline assembler*.

Uso dell'inline assembler

Il modo migliore per ottenere un programma il più efficiente e compatto possibile, consiste nello scriverlo manualmente in assembler. Di solito questo procedimento porta ad utilizzare il minor numero possibile di istruzioni ed anche il minor numero di risorse. Nessun compilatore (attualmente) è capace di raggiungere un simile livello di ottimizzazione. Anche se viene scelto di utilizzare il linguaggio C per scrivere il programma, è comunque possibile utilizzare l'assembler e le sue potenzialità per scrivere soltanto le parti più lente o computazionalmente più pesanti del programma. Queste parti possono essere inglobate nel codice C in modo molto semplice: è sufficiente infatti utilizzare la keyword `asm`, seguita dal blocco di codice assembler, come mostrato nell'esempio seguente:

```
void SetVideoMode(void)
{
    asm
    {
```

```
      mov ax,0x13
      int 0x10
    }
}
```

Questa funzione ha lo scopo di richiamare la modalità video 0x13 (grafica) su un PC (attenzione: i sistemi operativi più recenti potrebbero non gradire una simile operazione!). Il codice assembler è costituito soltanto da due istruzioni per il processore x86 ed il tutto è stato racchiuso in una funzione. Quest'ultimo particolare non è strettamente necessario, anche se in alcuni casi può facilitare un eventualmente passaggio di parametri. Nell'uso dei blocchi di codice assembler è buona norma salvare il valore dei registri su cui si andrà ad operare per ripristinarlo all'uscita del blocco (il compilatore non lo fa automaticamente). Si ricorda che l'uso del codice assembler, essendo strettamente specifico per una certa macchina, rende l'intero codice difficilmente portabile, tuttavia proprio la specificità dell'hardware può essere una delle ragioni principali per utilizzare l'inline assembler.
Specifiche caratteristiche peculiari di alcune architetture (ad esempio la presenza di alcuni tipi di coprocessori) non sono normalmente sfruttate dai compilatori ed è quindi necessario scrivere manualmente le funzioni che le utilizzino per accelerare l'esecuzione. Questo può essere fatto spesso solo in assembler. In questi casi è consigliabile porre queste funzioni in un modulo a parte, in modo da poterlo facilmente sostituire o modificare qualora venisse cambiato l'hardware. Va ricordato infine che la sintassi dell'inline assembler può varaire leggermente da un compilatore ad un altro, quindi è sempre consigliabile consultare la guida prima di utilizzare questa caratteristica.

Costrutti swhitch

Il costrutto `switch` è utilizzato di solito quando in un programma è necessario scegliere quale sezione di codice eseguire in base al valore di un parametro. Molti compilatori traducono questo costrutto con delle istruzioni di confronto e salto condizionato, poste sequenzialmente prima di ciascuno dei casi elencati. Questo implica che prima di raggiungere l'ultimo caso devono essere eseguite tutte le istruzioni di confronto e salto relative agli altri casi, con un notevole incremento del tempo di esecuzione. Per limitare questo effetto è possibile procedere in diversi modi. Un primo accorgimento è quello di controllare per primi i casi che si verificheranno più frequentemente (con più probabilità) quindi tutti gli altri. In questo modo i casi più frequenti richiederanno l'esecuzione di un numero minore di controlli e verranno eseguiti in minor tempo. È anche possibile sostituire il costrutto `switch` con l'istruzione goto ed una "tabella dei salti" (questo metodo risulta decisamente più complesso da mettere in pratica in generale).

Ulteriori consigli

Un'altra regola da tenere sempre presente per scrivere un codice molto efficiente è evitare assolutamente l'uso di tecniche di ricorsione. Questa infatti comporta un uso massiccio dello stack e quindi incrementa notevolmente il numero di accessi in memoria. Inoltre proprio per il fatto che lo stack può crescere molto, potrebbero verificarsi casi di *stack overflow*, cioè esaurimento dello spazio ad esso riservato.
Questo è ancora più probabile quando con le funzioni ricorsive vengono utilizzate le interruzioni. Anche le stesse interruzioni, in alcuni casi possono appesantire l'esecuzione del codice. Di solito infatti le routine di servizio delle interruzioni salvano il contenuto dei registri prima di iniziare le elaborazioni. Questo "*context switch*", se ripetuto frequentemente può costituire un notevole *overhead*. Nei casi in cui vi siano dei "treni" di interruzioni molto fre-

quenti, ma limitati nel tempo, può essere conveniente gestirle attraverso un polling, magari periodico anziché continuo. Un'altra osservazione interessante riguarda i loop realizzati con i cicli `for`. Normalmente un ciclo del tipo:

```
for(i=0; i<10; i++) {…}
```

viene tradotto in codice macchina come:

```
- esegui istruzioni del ciclo
- incrementa i
- confronta i e 10
- se diverso ripeti loop
```

Per controllare la fine del ciclo viene eseguita un'istruzione di *compare* quindi un salto condizionato. Se il ciclo venisse scritto utilizzando un conteggio decrescente, cioè:

```
for(i=10; i=0; i--) {…}
```

il codice macchina risulterebbe seguente:

```
- esegui istruzioni del ciclo
- decrementa i
- se i diverso da zero ripeti il loop
```

In questo caso non è stato necessario utilizzare l'istruzione di compare, dal momento che la maggior parte dei processori dispongono di un'istruzione di salto condizionata da un flag di zero. Anche se si tratta di una sola istruzione in meno, questa è eseguita ad ogni iterazione, quindi il tempo "sprecato" è dato dal tempo richiesto per eseguire l'istruzione di *compare*, per il numero di iterazioni (solo 10 in questo caso). Quando possibile quindi è conveniente invertire la direzione del conteggio dei loop.

16. Tecniche di Debug

INTRODUZIONE

Anche se potrebbe sembrare strano, la fase di test e di debug del codice, soprattutto nel caso di progetti piuttosto complessi, può richiedere più tempo di quello necessario per scrivere il codice. Questo è dovuto al fatto che i problemi che in genere si riscontrano nel provare il programma possono essere vari e può essere difficile capire la loro origine per riuscire a risolverli. Per questi motivi la fase di debug del codice deve essere tenuta in grande considerazione, già dalle prime fasi di sviluppo del programma. Solo in questo modo è possibile limitare il suo impatto sui tempi di sviluppo (quindi sui suoi costi). A tal fine è anche fondamentale conoscere ed utilizzare gli strumenti ed i metodi più appropriati per portare a termine questo compito nonché avere una discreta esperienza per individuare gli errori dai sintomi riscontrati.

INDIVIDUARE GLI ERRORI

Una volta completato un programma, normalmente lo si sottopone a una serie di test per controllare se è in grado di eseguire le operazioni per cui è stato creato e per verificare che esse siano realizzate correttamente (secondo le specifiche) oltre a verificare che il programma sia relativamente tollerante a ingressi o condizioni di funzionamento non previste. Raramente questi test vengono superati subito, di solito si riscontrano dei problemi ed è necessario intervenire sul codice per risolverli.
I possibili problemi sono di due tipi: il programma può funzionare normalmente ma fornendo risultati errati, oppure il programma può bloccarsi in determinate condizioni o in maniera apparentemente casuale. Nel primo caso probabilmente il problema è dovuto ad un errore nella scrittura dell'algoritmo o nella gestione dei dati. Il secondo caso invece è il più insidioso, sia perché può dipendere da cause molto diverse (quindi può richiedere maggiori sforzi per essere risolto) sia perché i malfunzionamenti possono non verificarsi subito e restare nascosti per molto tempo prima di potere essere rilevati.

L'origine degli errori

Nella maggior parte dei casi è molto più difficile capire l'origine degli errori che non risolvere il problema. Per questo la prima cosa da fare è cercare di osservare o rendere osservabile l'errore, isolarlo (cioè capire quali fattori interni o esterni lo generano e quali invece sono

ininfluenti) e tentare di riprodurlo, in modo da assicurarsi di avere bene identificato i meccanismi che lo generano. Riuscire ad osservare l'errore e le condizioni in cui esso si verifica in molti casi è il presupposto fondamentale per potere risolvere il problema. Molti dei metodi e degli strumenti presentati di seguito infatti, hanno proprio lo scopo di consentire un'osservazione più diretta e controllata dell'esecuzione del programma.

Sintomi tipici

Alcuni tipi di errori hanno delle manifestazioni abbastanza caratteristiche, è possibile quindi in molti casi individuare il tipo di problema anche soltanto dai suoi effetti. Ad esempio alcuni tra gli errori più frequenti sono quelli che riguardano la gestione della memoria. In genere questi errori hanno conseguenze piuttosto disastrose e portano ad un blocco del programma e/o alla corruzione dei dati. Una prima categoria di errori di memoria è generata da un uso non corretto di funzioni che allocano la memoria senza poi liberarla. Non si pensi solamente alla funzione `malloc`, lo stesso effetto può verificarsi anche con funzioni apparentemente più "inoffensive", quando queste sono richiamate un grande numero di volte. In genere in questi casi la memoria si esaurisce, oppure vengono sovrascritte delle aree che contenevano dati, codice o addirittura lo stack, con conseguente blocco del programma. Ovviamente non c'è modo di rilevare questo problema fino a quando non si verifica un blocco apparentemente "casuale". Un altro errore di memoria simile, è quello generato dall'esaurirsi dello spazio dedicato allo stack (*stack overflow*). Nei sistemi più semplici (privi di sistema operativo) questa condizione non è evidenziata in nessun modo, l'unico effetto visibile è che i risultati improvvisamente divengono errati, il programma esegue operazioni non previste ed infine può bloccarsi. Anche l'uso scorretto dei puntatori può portare a conseguenze simili a quelle viste, anche se in genere più circoscritte. Se sono state fatte delle assegnazioni sbagliate o viene commesso qualche errore di sintassi, è probabile che venga letta o scritta un'area di memoria sbagliata. Questo può causare errori persistenti nei dati o il cattivo funzionamento del programma stesso (se viene sovrascritta una sua sezione). Quando invece il programma sembra funzionare correttamente, ma i risultati sono errati, chiaramente è stato commesso un errore nella scrittura dell'algoritmo. In molti casi, escludendo errori concettuali nell'implementazione dell'algoritmo, il problema risiede in un'assegnazione errata (variabile al posto di puntatore o viceversa, oppure uso di tipi di lunghezza errata), nel mancato uso di parentesi o nella chiamata a funzioni in cui si sono utilizzati dei parametri errati o usati impropriamente.

Alcuni errori molto frequenti, ma poco visibili, sono causati dall'uso scorretto di variabili globali o da una mancata inizializzazione esplicita delle variabili. In questo caso si possono verificare errori nei dati (dovuti ad una perdita di coerenza), comportamenti diversi dopo ogni reset o il blocco del programma in alcuni loop (se alle variabili vengono assegnati valori non previsti potrebbe non verificarsi mai la condizione di uscita).

Una volta individuato il tipo di errore, un buon metodo per isolare il problema è quello di provare ad escludere le sezioni di codice sospette (ad esempio commentandole, quando possibile). Escludendo prima sezioni più grandi (intere funzioni), poi via via sempre più piccole (blocchi di codice e istruzioni), è possibile localizzare con precisione l'origine del problema.

Prevenire gli errori

Sicuramente prevenire gli errori è molto più conveniente che trovarli e correggerli in seguito. Questo richiede uno sforzo aggiuntivo in fase di scrittura del codice, che non sempre si è propensi a spendere. Tuttavia esso risulta indispensabile, soprattutto nel caso di progetti molto grandi, le cui parti sono sviluppate in parallelo da più programmatori. In questo caso può risultare molto costoso, se non impossibile, procedere per tentativi una volta termina-

to lo sviluppo dell'intero programma. È quindi necessario assicurarsi che tutto il codice prodotto funzioni bene da subito. Per fare questo è necessario in primo luogo che il programma sia sviluppato secondo un approccio *top-down*, cioè scomponendo il programma stesso in un insieme di funzioni più piccole e semplici, che poi vengono unite per formare via via funzioni più complesse. Il vantaggio di questo approccio consiste nel fatto che è possibile testare in modo abbastanza completo le singole funzioni elementari quindi avere la certezza che funzioneranno quando usate in un programma più complesso. Ovviamente anche le interazioni tra le varie funzioni elementari devono essere testate, ma questo in genere risulta meno problematico.
Nella stesura del codice si dovrebbe cercare di seguire sempre uno stile pulito e leggibile, in questo modo sarà più semplice trovare degli errori, perfino a chi non ha scritto il codice in prima persona. Inoltre la chiarezza del codice aiuta ad evitare errori dovuti proprio alla confusione del testo (parentesi mancanti o chiuse male, operatori con precedenza sbagliata…). Anche l'uso di molti commenti facilita il debug, infatti gli errori possono essere individuati semplicemente verificando che il codice esegua le funzioni descritte in linguaggio naturale nei commenti stessi.
Per evitare problemi di ambiguità o potenziali errori, è sempre meglio abbondare con le parentesi, sia nelle espressioni aritmetiche, sia nel passaggio di parametri "composti" a funzioni o macro. Non tutti i compilatori infatti considerano nello stesso modo le precedenze ed i raggruppamenti delle operazioni, quindi il codice dovrebbe sempre essere il meno ambiguo possibile per evitare problemi. Una corretta indentazione invece aiuta a suddividere meglio il codice e ad evitare di commettere errori nella chiusura dei blocchi, soprattutto in presenza di sezioni lunghe e annidate. L'uso delle variabili globali dovrebbe essere limitato al minimo indispensabile ed in ogni caso si dovrebbe prestare attenzione ed evidenziare le sezioni di codice in cui il contenuto di queste variabili viene modificato. Inoltre tutte le variabili dovrebbero essere inizializzate esplicitamente all'avvio del programma, per evitare che dopo un reset si possano verificare incoerenze o comportamenti imprevisti.
Infine molta attenzione deve essere dedicata all'uso dei puntatori. È consigliabile verificare con cura le sezioni di codice che li utilizzano ed assicurarsi anche di gestire i casi in cui ad essi possano essere assegnati valori non validi (molte funzioni di libreria ad esempio restituiscono un puntatore a `NULL` in caso di errore e questo ovviamente non deve essere usato come puntatore valido).

METODI DI DEBUG

Quando un programma è stato completato e deve essere eseguito sulla macchina target si riscontrano subito alcune difficoltà: è difficile capire se il programma sta funzionando bene e sapere in ogni momento cosa viene eseguito e come. In molti casi non basta sapere solo se i risultati finali sono corretti, ma occorre anche seguire l'evoluzione del programma e le singole operazioni. Questo problema è particolarmente sentito quando si programmano sistemi embedded non dotati di interfacce utente, ma che interagiscono con segnali e periferiche molto velocemente ed in maniera poco visibile dall'esterno.
Per ricavare maggiori informazioni si possono utilizzare appositi strumenti, che verranno descritti di seguito. Tuttavia esistono dei metodi piuttosto "primitivi", che si rivelano comunque molto utili nel caso in cui non si disponga di mezzi più sofisticati. Uno di questi, utile a capire se il programma sta eseguendo correttamente le operazioni più critiche, è quello di

utilizzare una banale istruzione `printf` (o equivalente), per visualizzare su un display o tramite comunicazione seriale delle informazioni quali ad esempio il valore di una variabile o la chiamata ad una funzione. Con questo sistema si può realizzare ad esempio un semplice tracing dell'esecuzione del programma: basta includere in ogni funzione un'istruzione `printf` che visualizzi il nome della funzione stessa ed eventualmente alcuni suoi parametri, come mostrato nell'esempio seguente:

```
#include <stdio.h>

// - Prototipi -
int LeggiValore(void);
int Funzione1(void);
int Funzione2(int);

main() {
  int valore, passo;

  valore=0;
  passo=0;

  // - ciclo principale -
  switch(passo) {

    case(0):
      valore=LeggiValore();
      passo=1;
    break;

    case(1):
      passo=Funzione1();
    break;

    case(2):
      passo=Funzione2(valore);
    break;

    default:
      // *** DEBUG! ***
      printf("Condizione imprevista!\n");
  }
}


int LeggiValore(void){
  …
  // *** DEBUG! ***
  printf("Funzione: LeggiValore\n");
  …
}
```

```
int Funzione1(void){
  …
  // *** DEBUG! ***
  printf(“Funzione: Funzione1\n”);
  …
}

int Funzione2(int val){
  …
  // *** DEBUG! ***
  printf(“Funzione: Funzione2(%d)\n”, val);
  …
}
```

Come si può vedere dal codice, il flusso di esecuzione del programma è difficilmente prevedibile, soprattutto se il valore restituito dalle funzioni dipende da parametri legati all'hardware. L'uso delle `printf` permette di apprezzare in quale ordine vengono chiamate ed eseguite le funzioni. In altri casi può essere utile anche dichiarare alcune variabili globali ausiliarie per potere osservare da qualsiasi punto dei valori particolarmente significativi assunti da alcune variabili locali (nota: queste variabili globali non interferiscono con il funzionamento del programma, quindi non introducono “rischi” aggiuntivi).
Va notato che, se non si dispone di periferiche di output testuali, la stessa tecnica può essere applicata in modo ancora più essenziale utilizzando dei LED o dei segnalatori acustici presenti nel sistema, per rilevare il passaggio da un determinato punto del programma. Questa stessa tecnica si rivela particolarmente utile anche per comunicare all'esterno determinate condizioni in cui si trova il programma. Ad esempio lo stato di una linea di I/O può essere utilizzato per generare il segnale di trigger per un oscilloscopio o analizzatore logico o per eseguire precise misure del tempo di esecuzione di alcune routine (si porta alto il livello della linea all'inizio della funzione e lo si abbassa alla fine). Un buon metodo per assicurarsi che il programma stia funzionando bene è quello di utilizzare delle *asserzioni*. L'idea di base è la seguente: se il programma sta funzionando come voluto, in determinati punti devono essere verificate delle precise condizioni.
Questa tecnica può essere parzialmente automatizzata usando appositi strumenti, ma può essere anche implementata manualmente. È sufficiente infatti porre nei punti voluti delle istruzioni che verificano le condizioni richieste e forniscano un certo output, oppure interrompano l'esecuzione del programma se queste non sono verificate. In questo modo è relativamente facile stabilire il punto in cui si originano gli errori. Il codice seguente implementa la tecnica appena descritta:

```
int Divisione(int p, int q) {

    // *** DEBUG ***
    if (q==0) {
       printf(“Funzione Divisione: q=0!”);
       while(1){}
    }

    return p/q;
}
```

L'istruzione `while(1)` è utilizzata per fermare l'esecuzione del programma. La sua utilità in fase di debug risiede nel fatto che spesso, in presenza di errori, il sistema potrebbe perdere il controllo del flusso di esecuzione e corrompere i dati presenti in memoria o comunque sovrascriverli, impedendo di rilevare le condizioni in cui si è verificato l'errore. Al contrario, potere osservare lo stato del sistema nel momento in cui si è verificato l'errore può fornire molte informazioni sulle sue cause. È necessario a questo punto soffermarsi su un particolare. Il codice aggiunto durante la fase di debug può influenzare il programma, rallentando l'esecuzione e in ogni caso, comporta un incremento delle dimensioni del codice oggetto. È desiderabile pertanto eliminarlo una volta terminata la fase di debug. Per facilitare questo compito è necessario in primo luogo renderlo ben visibile, evidenziandolo almeno con dei commenti, come fatto sopra. Può essere utile anche impiegare dei *tag,* cioè delle parole chiave facilmente riconoscibili, che possono essere ritrovate con una ricerca automatica nel testo. Una soluzione migliore è quella di utilizzare le direttive del preprocessore in modo da rendere possibile una compilazione condizionale, come mostrato nel codice seguente:

```
int Divisione(int p, int q) {

  // *** DEBUG ***
  #ifdef DEBUG
  if (q==0) {
    printf("Funzione Divisione: q=0!");
    while(1){}
  }
  #endif

   return p/q;
}
```

In questo modo è possibile abilitare o disabilitare tutte le sezioni di codice appositamente inserite per il debug soltanto definendo o meno la macro `DEBUG` (cioè inserendo un `#define DEBUG` in uno dei moduli che ha visibilità massima).

STRUMENTI

Nei precedenti paragrafi sono stati illustrati alcuni metodi che possono essere utilizzati per eseguire il debug di un programma. Di seguito verranno analizzati alcuni strumenti che possono facilitare ed accelerare sensibilmente questo lavoro.

Source-level debugger e simulatori

Costituiscono il primo strumento da utilizzare dopo avere scritto il codice. Sono programmi che vengono eseguiti sull'host (spesso forniti assieme ai compilatori) e permettono di eseguire il codice scritto "simulando" il processore, talvolta anche alcune periferiche esterne ad esso collegate o addirittura l'intero sistema. In genere è possibile eseguire il codice per intero o in modalità passo-passo ed avere la completa visibilità dei registri del processore, di quelli di eventuali periferiche e della memoria di sistema (anch'essa simulata).
Alcune delle funzioni più potenti sono quelle legate ai *watch* ed ai *breakpoint*. I primi danno la possibilità di controllare costantemente il valore assunto da alcune variabili, i secondi per-

mettono invece di bloccare il programma quando si verificano determinate condizioni, anche molto complesse.
Utilizzando congiuntamente questi due strumenti è possibile seguire l'esecuzione del programma e rendersi conto subito di eventuali errori, della loro posizione e delle loro cause. Ad esempio impostando un breakpoint su una condizione collegata ad uno degli errori riscontrati e simulando il programma, la simulazione si bloccherà al verificarsi della condizione e sarà possibile risalire al punto del programma che ha originato la condizione di errore, controllare il valore delle variabili e dei registri o eseguire passo-passo il programma.
L'uso dei simulatori può quindi accelerare molto la fase iniziale di test e debug. L'accuratezza dei simulatori può variare molto: alcuni simulano soltanto l'esecuzione delle istruzioni, altri eseguono delle simulazioni dell'hardware accurate al singolo ciclo di clock quindi permettono di scoprire anche problemi che si manifestano a livello più basso.

Hardware debugger

Anche quando il codice è stato verificato tramite un simulatore, può succedere che si verifichino degli errori quando il programma funziona in un ambiente reale (magari a causa dell'interazione con i segnali reali esterni e con le loro temporizzazioni). In questi casi un aiuto può essere dato dai debugger hardware.
Essi sono degli strumenti (che devono essere supportati dal sistema) che consistono essenzialmente in tre elementi: un'interfaccia di comunicazione tra il target e l'host (RS232, JTAG…), un piccolo programma (monitor) che viene aggiunto al codice caricato sul sistema target assieme a questo ed un programma "front-end" eseguito sul computer host, che permette di inviare comandi al sistema target e di visualizzarne i dati, comunicando con il programma monitor. Alcuni sistemi non necessitano di un programma *monitor* poiché sono dotati di un dispositivo hardware apposito, che svolge le stesse funzioni (tra l'altro in maniera più efficiente e meno invasiva). Il debugger hardware permette di interrompere l'esecuzione del programma sul target e compiere operazioni simili a quelle descritte prima a proposito dei simulatori, cioè leggere e scrivere il valore dei registri o della memoria, di eseguire passo-passo il programma e di impostare dei semplici breakpoint. In genere l'interazione avviene con comandi impartiti manualmente del tipo "leggi il registro x", "leggi n byte a partire dalla locazione di memoria y", "scrivi b nella locazione z", etc. Dopo avere esaminato i valori di interesse ed averli eventualmente modificati, è possibile anche riprendere l'esecuzione del programma. I degugger sono uno strumento molto utile e potente per controllare l'esecuzione del programma sul sistema reale e per testare il corretto funzionamento delle periferiche del sistema (a cui è possibile accedere agendo sullo spazio di memoria o di I/O del processore) e la loro interazione con il programma. Per aumentare le possibilità di intervento in molti casi può essere utile impiegare con il debugger alcuni dei metodi software visti prima. In particolare bloccare il programma quando si verificano determinate condizioni è utile per potere leggere dei valori in quella particolare situazione. Uno dei debugger più noti è lo GNU Debugger (GDB), che fa parte della catena di tool GCC ed è disponibile per moltissimi processori e sistemi. Molti altri debugger imitano lo stile ed il funzionamento del GDB.

In-Circuit Debuggers/Emulators

Sono degli strumenti estremamente potenti, che ultimamente si stanno diffondendo molto, fino ad essere integrati anche in piccoli microcontrollori. Essi in pratica uniscono la funzionalità dei simulatori e dei debugger hardware, permettendo di avere una completa visibilità e controllabilità dell'esecuzione del codice sul sistema reale. Utilizzando un In-Circuit Debugger/Emulator (abbreviati spesso ICD o ICE) è possibile seguire in tempo reale (o

quasi) l'evoluzione del programma, monitorare costantemente il valore dei registri, il contenuto della memoria, la presenza di interruzioni e soprattutto è possibile impostare complessi breakpoint hardware, simili a quelli utilizzabili con i simulatori. Queste funzioni facilitano molto l'individuazione di errori o malfunzionamenti dell'intero sistema e permettono di capirne facilmente le cause. Ad esempio, se una funzione di allarme non viene richiamata quando dovrebbe, si potrebbe scoprire che i valori forniti dal convertitore A/D non superano mai la soglia impostata, a causa magari di un guadagno troppo basso nella parte analogica. Oppure per capire perché alcuni dati assumono valori errati, si può impostare un breakpoint con la condizione "blocca l'esecuzione in caso di scrittura alla locazione x" quindi si possono individuare facilmente le istruzioni che originano l'errore.

Uso di oscilloscopi ed analizzatori logici

Può sembrare strano, ma spesso per capire i motivi per cui un programma non funziona come dovrebbe, non è sufficiente eseguire il debug sul solo software, ma è necessario estenderlo anche all'hardware. In particolare in molti casi l'evoluzione del programma è fortemente influenzata dall'interazione con segnali esterni. Talvolta l'origine di alcuni malfunzionamenti è dovuta al fatto che alcuni segnali non hanno le caratteristiche previste, anche solo per brevi periodi di tempo. Per eseguire queste verifiche è necessario monitorare congiuntamente i segnali hardware e l'esecuzione del software. Per fare questo vengono utilizzati prevalentemente due strumenti: gli oscilloscopi e gli analizzatori logici.

I primi sono più indicati per seguire l'andamento di segnali analogici ed in particolare rilevarne le caratteristiche in corrispondenza dell'esecuzione di alcune routine. Questo è possibile inviando ad un canale dell'oscilloscopio il segnale analogico in questione e all'altro il livello di un piedino di I/O pilotato in modo da fornire un riferimento temporale. Questa tecnica può essere utilizzata anche per verificare che i segnali generati dal programma rispettano le temporizzazioni previste.

Quando invece occorre verificare l'andamento e lo stato di diversi segnali digitali, lo strumento più indicato è l'analizzatore logico. Esso può registrare i segnali presenti su diverse linee (anche 32 o più) durante un certo intervallo di tempo. La registrazione viene eseguita ciclicamente su una memoria e si interrompe quando si verifica una precisa condizione (evento di trigger), che può essere data da una certa combinazione presente sulle linee o dallo stato di alcune di esse. In questo modo è possibile non solo visualizzare le caratteristiche dei segnali precedenti l'evento di *trigger*, ma anche le loro temporizzazioni con estrema precisione. Anche in questo caso, se serve, un I/O libero può essere utilizzato per sincronizzare gli eventi o per fornire un riferimento. Utilizzando l'analizzatore logico si possono facilmente scoprire fenomeni come *glitch*, *contentions* e termporizzazioni errate. È anche possibile seguire le comunicazioni su un bus di sistema o su una linea I^2C per verificare la loro correttezza o scoprire che alcuni dispositivi non rispettano le temporizzazioni attese (ad esempio generano segnali di clock a frequenza variabile o con duty-cycle diverso dal 50%).

17. Gestione delle interruzioni

INTRODUZIONE

Normalmente l'esecuzione di un programma è strettamente sequenziale e, anche in presenza di salti o chiamate a subroutine, è sempre possibile seguirne o prevederne l'evoluzione. In altre parole un qualsiasi programma, soprattutto nel caso di sistemi embedded, è praticamente un loop infinito (il classico `while(1){…}`), che esegue ciclicamente tutte le operazioni necessarie al controllo ed alla gestione del sistema (figura 17.1).

In molti casi però questo tipo di funzionamento non risulta adeguato alla gestione di certi problemi. Spesso infatti è richiesto che al verificarsi di certi eventi esterni (più o meno imprevedibili), il programma risponda immediatamente, per poi tornare al suo normale funzionamento. Per ottenere questa caratteristica viene utilizzato il meccanismo delle *interruzioni* (*interrupt* in inglese), che è utilizzabile nella maggior parte di microprocessori e microcon-

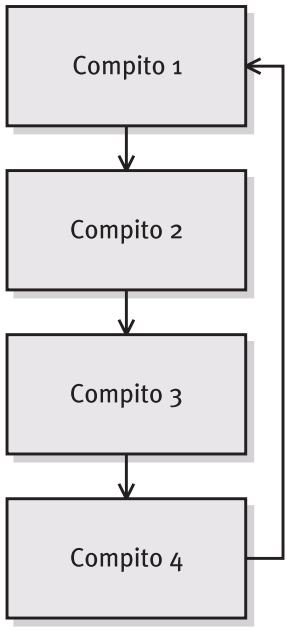


Figura 17.1
Struttura di un programma puramente sequenziale

trollori. Una interruzione è quindi un segnale esterno che provoca un temporaneo ed immediato cambiamento del flusso di esecuzione del programma.
Un programma che utilizza le interruzioni ha di solito una struttura diversa, in quanto oltre al loop principale, alcune funzioni sono svolte dalla (o dalle) routine di interruzione (figura 17.2). In questi casi quindi è necessario un progetto apposito e molto più accurato e questo è ancora più importante quando si utilizza un linguaggio ad alto livello (in assembler risulta relativamente più diretto e naturale).

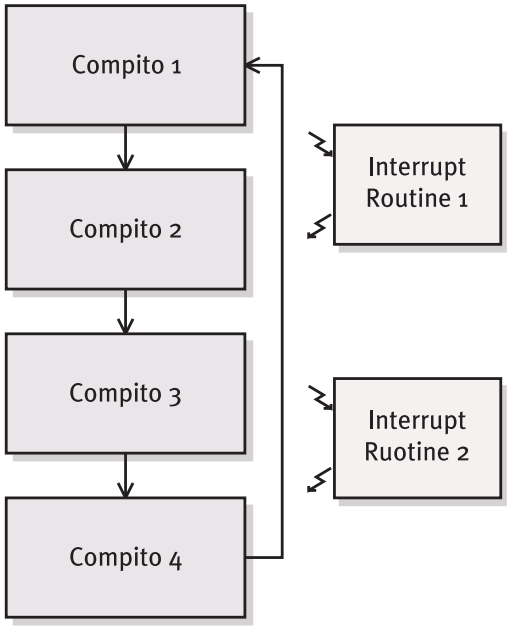


Figura 17.2
Struttura di un programma che utilizza le interruzioni

CARATTERISTICHE DELLE INTERRUZIONI

La principale caratteristica delle interruzioni è la loro imprevedibilità: possono essere generate in un qualsiasi momento, provocando l'interruzione del programma in un qualsiasi punto. Se il tutto è stato progettato accuratamente ci sarà un immediato salto alla routine di servizio delle interruzioni (*Interrupt Service Routine* o ISR), che svolgerà il compito che le è stato affidato. Una volta terminata la routine ISR, verrà ripresa la normale esecuzione del programma da dove era stato interrotto. Il funzionamento è simile a quello di una qualsiasi subroutine, solo che in questo caso la sua chiamata avviene in maniera imprevedibile in un qualsiasi punto del codice. L'altra caratteristica delle interruzioni è che devono essere prese in considerazione immediatamente dal processore ed essere soddisfatte nel più breve tempo possibile. In questo modo il loop principale non risente molto del tempo perso per eseguire la ISR. Alcuni esempi tipici dell'uso di interruzioni sono i seguenti:

1) Segnalazione di eventi esterni improvvisi: allarmi, pressione di pulsanti, superamento di alcune soglie da parte di certi dispositivi o sensori, etc. Usando questo approccio il microprocessore non avrà bisogno di controllare in continuazione questi parametri, ma riceverà una segnalazione solo quando sarà successo qualcosa che necessita attenzione;

2) sincronizzazione o temporizzazione: disponendo di un interrupt generato da uno o più timer, è possibile rendere regolare l'esecuzione di vari compiti o in generale sincronizzare l'esecuzione del programma con qualche evento o dispositivo esterno. Questo approccio è spesso usato nei sistemi operativi real-time per schedulare ed eseguire i vari processi.
3) sapere quando un dato è disponibile o una periferica è pronta, senza dovere attendere esplicitamente. In questo caso invece di utilizzare un loop per leggere lo stato di una periferica al fine di attendere che si liberi o fornisca un dato (tecnica denominata *polling*), è possibile continuare ad eseguire il programma e riprendere il lavoro solo quando verrà generata un'interruzione. Questa tecnica permette di sfruttare meglio le capacità di elaborazione del processore, evitando inutili attese. Un esempio tipico di questa tecnica è l'interrupt fornito da una UART alla ricezione di un carattere o la fine di una conversione da parte di un ADC.

È chiaro che sfruttando queste tecniche, il codice assume una struttura diversa da quella tipicamente sequenziale.

MECCANISMI D'INTERRUZIONE

Nella maggior parte dei microprocessori gli interrupt sono comandati dallo stato di alcune linee dedicate (indicate di solito come INT o IRQ), è sufficiente comandare opportunamente queste linee per richiamare la (o le) ISR. Nei microcontrollori invece le linee di interruzione sono spesso associate a piedini di I/O ed il loro funzionamento è programmabile. Non solo, anche le periferiche interne possono generare interruzioni se abilitate (UART, ADC, DMA, timer…). In entrambi i casi è necessario capire quale periferica ha generato l'interruzione. I metodi più comuni sono i seguenti:
1) Non viene fornita questa informazione durante l'interruzione, quindi la ISR dovrà andare a leggere i registri delle varie periferiche (eseguendo un *polling*) per capire chi ha generato l'interruzione;
2) assieme al segnale di interruzione viene fornito (in qualche modo) un *vettore d'interruzione*, che indica quale periferica lo ha generato;
3) viene utilizzato un controllore d'interruzione esterno o interno, il quale genera il vettore.

Utilizzando il polling o talvolta anche impiegando un controllore esterno, la ISR è quasi sempre una sola e provvede a "smistare" l'interruzione alle funzioni corrette. Utilizzando invece un vettore d'interruzione, il processore ricava da questo un indirizzo di memoria in cui si trova la ISR. In questo caso ci saranno approssimativamente tante ISR quante sono le periferiche interrompenti. Spesso gli indirizzi ricavati sono molto vicini tra loro e si trovano nell'area iniziale della memoria, quindi contengono solo istruzioni di salto alle funzioni ISR vere e proprie. Quest'area di memoria viene chiamata *tabella dei vettori d'interruzione*. Gli interrupt possono essere abilitati e disabilitati via software, sia singolarmente che nella loro totalità. Su alcuni sistemi però può esistere un particolare ingresso, chiamato NMI, acronimo di *Non Maskable Interrupt* (interrupt non mascherabile), che non può essere disattivato in alcun modo via software. Questo tipo di interrupt, quando presente, viene utilizzato per segnalare situazioni critiche per il sistema (errori, allarmi gravi, spegnimenti improvvisi…), per fornire un segnale di temporizzazione fisso o per richiamare periodicamente delle routine di alcuni sistemi operativi (che in questo modo mantengono il controllo rispetto al software utente).

Le interruzioni possono essere generate dai livelli logici oppure dalle transizioni presenti sulle rispettive linee. Nel primo caso il dispositivo che genera l'interruzione mantiene la linea nello stato "attivo" fino a quando la CPU non segnala che l'interruzione è stata soddisfatta e solo a quel punto la linea viene rilasciata. Spesso questa segnalazione deve essere fatta esplicitamente dalla ISR, altrimenti, una volta terminata la routine, essa verrebbe richiamata di nuovo indefinitamente! Nel caso di interruzioni sensibili ai fronti può invece verificarsi la situazione opposta: se in un determinato istante le interruzioni sono disabilitate e la CPU non cattura la transizione via hardware, la richiesta può essere persa! Un altro aspetto da considerare è quello della priorità tra le interruzioni. Se esistono diverse sorgenti, è necessario ordinarle secondo una priorità crescente, in modo da stabilire un comportamento nel casi in cui più interruzioni si verificano contemporaneamente. Indipendentemente da tutte le varianti appena viste, occorre notare che quando si verifica un'interruzione, per potere saltare alla ISR e riprendere poi l'esecuzione normalmente, occorre salvare il *contesto di esecuzione*. In primo luogo deve essere salvato il Program Counter (cioè l'indirizzo dell'istruzione che stava per essere eseguito) ed eventualmente altri registri sensibili, come ad esempio l'accumulatore ed il registro di stato. La ISR infatti potrebbe modificare il contenuto dei registri, quindi riprendendo la normale esecuzione del programma, i valori potrebbero differire da quelli precedenti all'interruzione. In genere il Program Counter viene salvato automaticamente dal processore stesso nello stack, gli altri registri devono essere salvati manualmente all'inizio della ISR o usando lo stack, oppure cambiando il set di registri utilizzati, quando possibile. I valori dovranno essere poi ripristinati prima di uscire dalla ISR.

GESTIONE DELLE INTERRUZIONI IN C

L'ANSI C non prevede funzioni native o di libreria per la gestione delle interruzioni, pertanto è necessario affidarsi a librerie o estensioni proprietarie fornite dai vari compilatori. Non solo, queste estensioni possono differire molto da un prodotto ad un altro, vista anche la grande varietà di meccanismi offerti dalle varie architetture target. Nella maggior parte dei casi il compilatore permette di utilizzare delle keyword o dei nomi appositi per designare una normale funzione come ISR. In molti casi ad esempio è sufficiente chiamare la funzione ISR con un nome riservato, oppure aggiungere le parole `interrupt`, `int_hadler` o simili prima o dopo la dichiarazione della funzione. Ad esempio:

```
/* MikroC per PIC */
void interrupt(void);


/* GCC (alcune versioni) */
void int_UART(void) __attribute__ ((interrupt_handler));
```

Nei casi in cui il processore supporti interruzioni vettorizzate, possono essere specificate più ISR e per ciascuna è necessario aggiungere anche qualche informazione sul vettore, ad esempio:

```
/* ZDS-II per Z8Encore! */
void interrupt mia_ISR(void)
{
```

```
  SET_VECTOR(INT0,int0_handler);
  …
}


/* SDCC per 8051 */
void timer_isr (void) interrupt 1 using 1
```

In quest'ultimo caso è stato specificato anche il set di registri alternativi da utilizzare per la ISR. Alcuni compilatori, anche per CPU non dotate di vettorizzazione, permettono di trattare le interruzioni come se fossero vettorizzate, cioè permettono di scrivere ISR separate per diverse sorgenti. Questo è possibile perché l'ISR vera e propria viene creata dal compilatore e svolge la funzione di "smistare" le interruzioni:

```
/* CCS per PIC */
#INT_AD
void adc_handler(void){
…
}

#INT_RTCC NOCLEAR
void rtc_isr(void){
…
}
```

Le due ISR mostrate sono quelle dell'ADC e del timer di un PIC. Notare che nel secondo caso è stata utilizzata la keyword aggiuntiva `NOCLEAR`, che specifica di non resettare il flag di interruzione della periferica (quindi non specificare che l'interrupt è stato servito). In genere i compilatori dispongono di diverse keyword come queste, per accedere ai meccanismi di più basso libello. Non è possibile entrare nei dettagli delle varianti relative a diversi compilatori ed architetture e si rimanda per questo al manuale dello specifico compilatore utilizzato. In tutti i casi la funzione ISR non prende in ingresso alcun valore tanto meno restituisce un risultato. Inoltre va precisato che molti compilatori traducono automaticamente la funzione ISR aggiungendo il codice necessario per salvare il contesto. I registri salvati ed il modo in cui sono salvati dipendono dal compilatore.

Uso delle interruzioni

Per chiarire meglio le possibilità e l'utilizzo delle interruzioni si considerino inizialmente due casi estremi: un programma sequenziale, che non usa le interruzioni ed un programma basato *solo* su interruzioni. Come esempio si consideri un orologio a LED, dotato di un tasto per commutare tra le 4 diverse funzioni ed un'interfaccia seriale per potere leggere o impostare i valori da un terminale remoto. Un tipico programma sequenziale sarà formato da un loop infinito dentro il quale saranno richiamate ciclicamente le varie funzioni di gestione (figura 17.3). Ad ogni ciclo verrà fatto un polling sul timer, sullo stato del tasto, sulla UART e verrà aggiornato il display di conseguenza.

```
/* -- nel main -- */
…
while(1) {
```

```
   t=Leggi_timer();
   ...
   p=p+Leggi_stato();
   p&=3;
   …
   c=Leggi_UART();
   …
   Aggiorna_display(p);
}
```

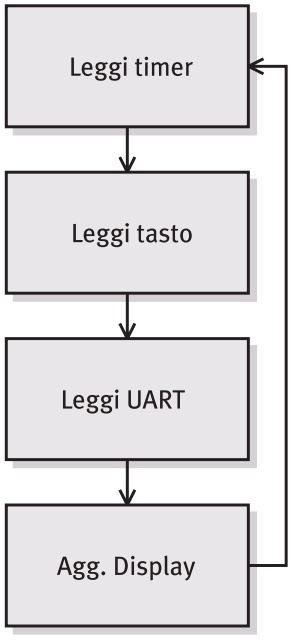


Figura 17.3
Programma sequenziale per la gestione dell' orologio a LED

Se le routine sono ben progettate il tutto potrebbe funzionare senza problemi, ma sono sufficienti piccoli ritardi imprevisti per perdere un carattere ricevuto dalla UART, oppure non accorgersi di una breve pressione del tasto. Inoltre un codice scritto in questo modo risulterà difficilmente modificabile o aggiornabile.

Utilizzando la filosofia diametralmente opposta si potrebbe pensare di utilizzare un loop principale "vuoto" e di gestire il sistema soltanto tramite le funzioni di interruzione. In pratica il sistema resta inerte fino a quando non arriva un interrupt (figura 17.4). Ogni 0.5 secondi ci sarà un interrupt del timer, che servirà per aggiornare il puntino lampeggiante e dopo due passaggi, l'orario visualizzato; alla pressione del tasto si aggiornerà lo stato della visualizzazione, alla ricezione di un carattere dalla UART verrà eseguita la funzione di comunicazione. Tutto questo in maniera automatica:

```
#INT_RTCC
void Timer_int(void){
…
```

```
}

#INT_BUTTON
void Tasto_int(void){
…
}

#INT_RDA
void UART_int(void){
…
}


main() {
  …
  while(1){};
}
```

Come già detto è possibile utilizzare un simile approccio anche su processori che non hanno interrupt vettorizzati, basta che la ISR principale richiami le funzioni appropriate. Questo approccio è simile a quello ad "eventi" che viene utilizzato nella gestione delle interfacce grafiche a finestre di diversi sistemi operativi.
Più realisticamente un programma avrà quasi sempre un loop principale in cui vengono eseguiti i compiti a bassa priorità e delle routine di interruzione per gestire i compiti temporalmente più critici. Rimangono comunque da analizzare alcuni aspetti poco intuitivi riguardo l'uso delle interruzioni.

All'interno di una ISR…

Alla chiamata di una ISR viene salvato automaticamente il Program Counter ed i registri principali (accumulatore, status register, etc…). In genere viene data la possibilità di scegliere un set di registri alternativi, se esiste o il compilatore stesso fa in modo da non interferire con quelli precedentemente utilizzati.

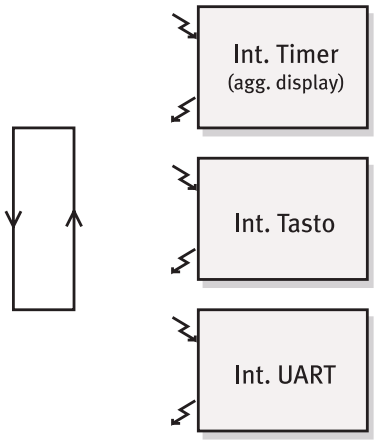


Figura 17.4
Programma basato solo su interruzioni per la gestione dell' orologio a LED

La prima cosa da fare dentro la ISR, se non la fa automaticamente il compilatore, è disabilitare le interruzioni. Se il processore utilizzato supporta gli interrupt vettorizzati, allora è stata già richiamata la funzione appropriata, altrimenti è necessario eseguire una lettura di uno o più registri per capire quale periferica ha generato l'interruzione. A questo punto è necessario dare una conferma dell'avvenuta interruzione resettando il o i bit relativi alla periferica considerata (situati su un registro della periferica stessa o del controller delle interruzioni). Dopo avere svolto i compiti richiesti, sarà sufficiente riabilitare le interruzioni (se necessario) ed uscire dalla funzione. Un esempio di ISR relativo ad un microcontrollore PIC può essere la seguente:

```
void interrupt(void) {
  if (TestBit(INTCON, T0IF)) {
    counter++;
    TMR0 = 127;
    ClearBit(INTCON, T0IF);
  }
  else if (TestBit(INTCON, RBIF)) {
    counter=0;
    TMR0 = 127;
    ClearBit(INTCON, RBIF);
  }
}
```

In questo caso si è supposto che siano abilitati soltanto due sorgenti d'interruzione: il timer TMR0 e il cambiamento sui bit della porta B e che il compilatore si occupi del salvataggio dei registri e della disabilitazione delle interruzioni. Dal momento che i PIC non hanno un meccanismo di vettorizzazione, è necessario fare un polling: dapprima viene testato il timer, successivamente la porta B. Il registro INTCON contiene lo stato delle interruzioni e rivela quale delle due si sia verificata.
Nel caso di overflow del timer, quindi viene incrementata la variabile counter, caricato nuovamente il timer e viene cancellata l'interruzione pendente. Nell'altro caso la variabile viene azzerata. I bit T0IF e RBIF del registro INTCON sono quelli corrispondenti alle sorgenti considerate, mentre le macro `TestBit` e `ClearBit` testato e resettano un determinato bit del registro specificato. Notare che, per come sono scritte le due condizioni, esse sono mutuamente esclusive.
Non sempre i compiti richiesti per soddisfare un'interruzione sono eseguite dentro la ISR, questo perché potrebbe risultare troppo lento e potrebbero essere perse altre interruzioni. Per questo vengono talvolta utilizzate funzioni esterne, che però non possono essere chiamate direttamente dalla ISR (per problemi di tempi di esecuzione e/o di stack), ma devono essere eseguite dopo che la ISR è terminata. Per fare questo vengono passati dei parametri al loop principale come descritto nei prossimi paragrafi.

Passaggio di dati

Poiché le ISR non accettano ne restituiscono dati, per comunicare con il resto del programma si fa uso di flag o di variabili globali, settate dalla ISR e lette dalle routine del loop principale (o viceversa). Ritornando all'esempio dell'orologio a LED visto prima, la parte dell'ISR che gestisce la pressione del tasto potrebbe aggiornare solo la variabile relativa allo stato di visualizzazione:

```
stato=(stato++)&3;
```

In questo modo alla pressione del tasto la variabile `stato` assume i valori 0, 1, 2, 3, 0, 1, etc… La funzione di visualizzazione presente nel loop principale terrà conto del valore della variabile per scegliere i dati da visualizzare. Lo stesso ragionamento può essere fatto per la gestione della UART: nella ISR relativa i dati ricevuti verranno semplicemente scritti su un buffer, sarà poi la routine apposita del loop principale che decodificherà i comandi ed invierà le risposte. Dei semplici flag possono essere invece utilizzati per comunicare al loop principale il fatto che sia avvenuta una certa interruzione o che deve essere eseguita una funzione. Nel loop principale, tra i vari compiti da eseguire ci sarà un'istruzione tipo:

```
if (FLAGB == 1) {
  mia_funzione();
  FLAGB=0;
}
```

Sezioni critiche

Si supponga che nel loop principale di un programma sia contenuta la seguente riga di codice:

```
val=val*cost+val;
```

se durante l'esecuzione di questa riga di codice arriva un'interruzione e la ISR legge o scrive la variabile `val`, il risultato ottenuto sarà molto probabilmente errato. Questo succede perché l'esecuzione della riga riportata sopra non è istantanea, ma è eseguita in alcuni passi (cioè è scomposta in diverse istruzioni macchina). Se avviene un interruzione proprio nel mezzo di questi passi, da un lato la ISR leggerà un valore errato della variabile (un valore intermedio assunto durante il calcolo), dall'altro lato al ritorno la variabile sarà stata modificata e quindi il calcolo sarà concluso utilizzando in parte un valore diverso dall'originale! In questi casi si dice che il codice in questione è una sezione critica. Esistono moltissimi casi in cui si possono avere delle *sezioni critiche*: oltre che nella manipolazione di variabili, anche in determinate routine complesse o legate alla comunicazione. Per evitare problemi con le sezioni critiche occorre renderle *atomiche*, cioè fare in modo che, una volta iniziate, esse vengano terminate prima di cedere il controllo alla ISR. Per fare questo è sufficiente disabilitare le interruzioni prima di iniziare e riabilitarle alla fine. Alcuni compilatori hanno anche delle keyword apposite per specificare le sezioni critiche o comunque per disabilitare ed abilitare le interruzioni. Va notato che perfino operazioni banali come un'assegnazione tra grandezze di tipo `long`, eseguita su un microprocessore ad 8 bit rappresenta una sezione critica, in quanto per completare l'assegnazione occorrono 4 istruzioni di movimento ad 8 bit e nel mezzo di queste possono verificarsi delle interruzioni. Per questo motivo, più che utilizzare la soluzione proposta prima, si preferisce di solito evitare di modificare nell'ISR le variabili modificate dal loop principale e viceversa. Tuttavia per alcuni algoritmi o procedure piuttosto articolate (in cui i dati non devono cambiare nel corso dell'esecuzione), rimane necessario rendere atomiche le sezioni interessate. In fase di progettazione e scrittura del codice occorre prestare molta attenzione ai potenziali errori originati da sezioni critiche, perché può essere difficilissimo riuscire ad individuarli una volta che il programma è in esecuzione, dal momento che si presentano in modo sporadico, casuale e poco ripetibile.

Interrupt multipli

Con alcuni processori che supportano le interruzioni vettorizzate non sussistono problemi a ricevere altre interruzioni durante l'esecuzione di una ISR: quello che succede è un salto ad un'altra ISR, conservando il contesto e quando questa sarà terminata si ritornerà ad

eseguire la precedente. Si possono verificare anche diverse interruzioni annidate e tutto può funzionare correttamente. Dal punto di vista pratico può però sorgere qualche problema: innanzitutto i vari contesti devono essere memorizzati nello stack che di solito ha una dimensione limitata rendendo probabile un errore di stack overflow. In alcuni processori, come i PIC, lo stack è implementato via hardware ed ha una profondità molto limitata (oltre a non essere manipolabile dall'utente) quindi questa eventualità è molto più probabile. Anche cercare di limitare via software il numero di interrupt serviti può risultare poco pratico e sicuro. In molte applicazioni comunque si può stimare con precisione il numero massimo di interruzioni che possono arrivare, quindi accettare più interruzioni è un processo relativamente innocuo. In altri casi, di solito quando è presente un controllore delle interruzioni, è possibile ordinare le interruzioni per priorità: in questo modo una ISR può essere interrotta soltanto da un segnale che ha priorità più elevata. Questo meccanismo limita il numero di interruzioni annidate e permette di ottimizzare i tempi e le prestazioni. Va notato che la priorità in qualche caso può essere anche programmata via software, per cui può essere resa dinamica in funzione delle esigenze.
In altri casi si ha la segnalazione di un interruzione "cumulativa" che può essere associata a diverse periferiche in attesa di risposta. Sarà la ISR a rendersi conto di questo leggendo un opportuno registro, decidendo quale soddisfare prima. Se non tutte le interruzioni pendenti sono state soddisfatte, all'uscita della ISR sarà generata una nuova interruzione e così fino a quando tutte le periferiche non saranno state soddisfatte. L'esempio riportato prima rientra proprio in questo caso e implementa anche via software un meccanismo di priorità tra le interruzioni: le due condizioni infatti sono scritte in modo che solo una alla volta sarà soddisfatta ed in particolare sempre quella scritta prima.

QUANDO USARE LE INTERRUZIONI?

Le interruzioni sono il più stretto punto di collegamento tra hardware e software, per cui scegliere se, quando ed in che caso utilizzarle, non è una questione che riguarda solamente il software, ma interessa la progettazione del sistema nella sua interezza. È possibile scrivere un software interamente *interrupt driven* (basato solo sulle interruzioni) solo se l'hardware fornisce il giusto supporto. D'altro canto pur avendo a disposizione moltissime interruzioni si può scegliere di scrivere un software che non ne faccia uso (quindi puramente sequenziale). In alcuni casi l'uso delle interruzioni è indispensabile per ottenere buone prestazioni. Ad esempio un sistema operativo real-time dovrebbe essere basato su un segnale di temporizzazione regolare e stabile per schedulare i vari task. Oppure qualsiasi evento critico dovrebbe avere una risposta immediata da parte del software.
Si può notare anche che in alcuni casi utilizzare le interruzioni può semplificare molto la scrittura del software, semplificando il loop principale, evitando di dovere progettare in maniera accurata lo scheduling e le temporizzazioni e rendendo più modulare il programma. Ma anche l'hardware può risultare semplificato in qualche caso, infatti è possibile delegare più compiti di controllo al software. In qualche caso però l'uso delle interruzioni può anche presentare degli svantaggi. Un esempio si ha nel caso di processori in cui devono essere salvati molti registri: questo può rappresentare uno zoccolo di tempo sprecato ad ogni interruzione, che diviene rilevante se gli eventi si succedono con grande frequenza. In questi casi può risultare vantaggioso perfino l'uso del polling. L'altro svantaggio notevole nell'uso degli interrupt è la maggiore difficoltà nel debug del software. In fase di simulazione e di test molti strumenti fondamentali, come gli hardware debugger e molti degli

ICD/ICE, non funzionano correttamente in presenza di interrupt. Anche durante il normale funzionamento del sistema, individuare degli errori generati dagli interrupt risulta molto difficile, in quanto si ha un grado di osservabilità e di controllabilità notevolmente ridotto rispetto al caso di semplice codice sequenziale.

18. Sistemi operativi

INTRODUZIONE

Fino ad ora sono stati considerati programmi singoli ed autosufficienti, cioè programmi che vengono eseguiti dal processore o microcontrollore e che hanno il completo controllo del flusso di esecuzione e dell'hardware del sistema. Moltissimi software per piccoli sistemi embedded sono scritti in questo modo: tutti i compiti sono gestiti da un loop principale ed in parte dalle routine di interruzione. Se però i compiti da svolgere sono molti, le loro interazioni complesse e le temporizzazioni abbastanza critiche, diventa molto difficile gestirli nel modo già visto. Si pensi ad esempio ad un sistema di controllo industriale che deve gestire, oltre ad un complesso e delicato processo, anche una comunicazione di rete basata sul protocollo TCP/IP, un'interfaccia utente grafica e le relative periferiche di input ed output. In questi casi l'uso di un sistema operativo (*Operating System*, in breve OS) potrebbe semplificare molto lo sviluppo del software, renderlo anche più modulare, efficiente e robusto.
Al contrario di quello che si potrebbe pensare, un sistema operativo non è necessariamente uno strumento complesso (come quelli utilizzati sui personal computer). Esistono sistemi operativi molto semplici che possono essere eseguiti perfino su microcontrollori ad 8 bit! Una volta compresi i principi su cui si basa il loro funzionamento non è neanche eccessivamente difficile scriverne uno secondo le proprie esigenze.
Un OS svolge diverse funzioni: in primo luogo permette di eseguire "contemporaneamente" più programmi o di svolgere più compiti, in secondo luogo si occupa della gestione di basso livello dell'hardware, mettendo a disposizione dei programmi un'interfaccia più astratta e fornendo anche diversi strumenti utili per facilitare la scrittura. Le caratteristiche e le funzioni messe a disposizione dipendono dal particolare sistema operativo e possono variare moltissimo da un prodotto all'altro.

COME FUNZIONA UN SISTEMA OPERATIVO

Come già detto esistono sistemi operativi complessi che offrono funzioni avanzate, come ad esempio il supporto per le periferiche di archiviazione di massa, un file system, protocolli di comunicazione (stack TCP/IP) ed il supporto per le periferiche grafiche. Ovviamente sistemi operativi di questo tipo risultano molto più costosi ed esigenti in termini di risorse ed in certi casi più complessi da utilizzare. Tutti i sistemi operativi comunque sono basati sugli stessi concetti e meccanismi di base, che verranno descritti in questo capitolo.

Task

Uno dei vantaggi principali offerti da un sistema operativo è la possibilità di eseguire simultaneamente routine diverse ed indipendenti. Ciascuna di queste routine viene definita *task* (in inglese *compito*). Ovviamente l'esecuzione dei task non è realmente simultanea, ma essi vengono eseguiti a turno condividendo così le risorse dell'unica CPU. È il sistema operativo che decide, momento per momento, quale task deve essere eseguito e quale restare in attesa. I task possono alternarsi molto velocemente quindi il risultato è simile a quello che si avrebbe se venissero eseguiti davvero in concorrenza. I vari task sono programmi del tutto indipendenti tra loro ed a ciascuno sono associati contesti e parametri diversi. Ad esempio ciascun task ha un suo stack, un suo identificativo per distinguerlo dagli altri ed una sua priorità di esecuzione assegnata dall'utente.
Quando un task viene momentaneamente sospeso in favore di un altro, vengono conservate tutte le informazioni necessarie per potere riprendere l'esecuzioni nelle stesse condizioni in cui la si è lasciata. Questo meccanismo è simile al salvataggio del contesto che avviene durante le interruzioni: verrà salvato ad esempio il Program Counter, lo status register, i vari registri e verrà cambiato lo stack.

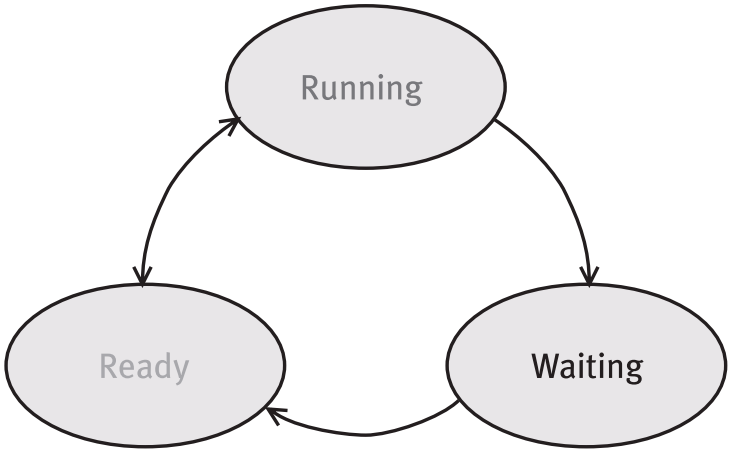


Figura 18.1
Stati di eseuzione dei task e loro transizioni

Ogni task può trovarsi in diversi stati di esecuzione: può essere correntemente in esecuzione (*running*), bloccato in attesa di qualche evento (*waiting*) o pronto per l'esecuzione, ma non eseguito in quel momento (*ready*). La figura 18.1 rappresenta questi stati e le loro relazioni. Un task si trova nello stato di attesa (*waiting*) se sta attendendo che si verifichi un evento esterno o se ha l'esigenza di restare in "pausa" per un certo periodo di tempo. Fintanto che un task si trova nello stato di attesa, il sistema operativo non lo considererà ai fini dell'esecuzione, ma controllerà se si sono verificate le condizioni per riportarlo nello stato *ready* (si è verificato l'evento esterno atteso o è trascorso il tempo di attesa specificato). I task nello stato *ready* sono quelli che di fatto si contendono l'esecuzione e quindi lo stato *running* (in cui ovviamente si può trovare un solo task alla volta). La scelta del task da eseguire, tra quelli *ready*, dipende dagli algoritmi di scheduling utilizzati dall'OS.

Lo scheduler

Il cuore di ogni sistema operativo è lo *scheduler*, è lui infatti a gestire la scelta dei task da eseguire, a cambiarne lo stato o a stabilire se un task deve essere sospeso in favore di un altro a priorità maggiore. L'algoritmo impiegato dallo scheduler per stabilire quale task eseguire in un dato momento, determina gran parte delle caratteristiche dell'intero sistema operativo. Esistono diversi possibili algoritmi di scheduling (come il "*first in first out*" o lo "*shortest job first*"), ma quello che più si avvicina alla descrizione che è stata fatta fino ad ora e che offre più possibilità, è il *round robin* con le sue varianti. In questo caso i task nello stato *ready* sono eseguiti circolarmente, uno dopo l'altro. Utilizzando un approccio cooperativo (detto anche *non-preemptive* o senza prelazione) allora sarà il task in esecuzione (e solo lui) a decidere di sospendersi quando avrà completato il suo compito, passando nello stato *waiting* o *ready* e dando così l'opportunità agli altri di essere eseguiti (figura 18.2). Prima di essere attivato di nuovo occorrerà che tutti gli altri task *ready* siano stati eseguiti una volta. Questo algoritmo dà a tutti i task uguale opportunità di essere eseguiti, ma non offre garanzie sui tempi necessari per eseguire un "giro" completo, quindi su quanto tempo sarà necessario prima che un determinato task sia eseguito di nuovo. Dal momento che il passaggio da un task ad un altro avviene via software, questo tipo di scheduling può essere utilizzato anche senza fare ricorso alle interruzioni. Un approccio diverso è quello di uno scheduling di tipo *preemptive* (con prelazione), in questo caso lo scheduler può interrompere forzatamente l'esecuzione di un task per eseguirne un altro, quando si verificano le condizioni opportune. Normalmente lo scheduling preemptive è associato all'uso di priori-

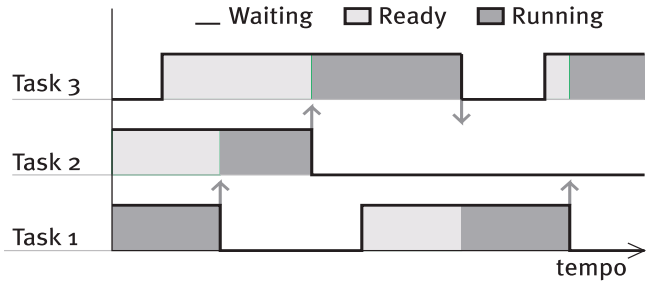


Figura 18.2

Esempio di scheduling cooperativo

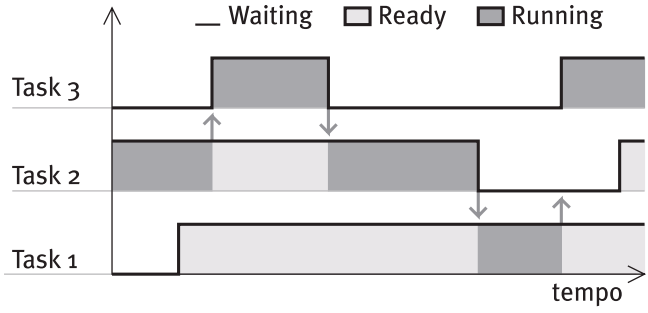


Figura 18.3

Esempio di scheduling preemptive (i task sono ordinati per priorità crescente)

tà tra i task: se in seguito ad un evento un task a priorità alta passa dallo stato *waiting* a *ready* ed in quel momento è eseguito un task a priorità più bassa, allora questo viene sospeso (cioè passerà da *running* a *ready*) in favore del primo (figura 18.3). Quando il task a priorità più alta tornerà allo stato *waiting* allora l'esecuzione del secondo potrà riprendere (sempre che non ci siano altri task a priorità più alta *ready*). Se tutti i task hanno priorità uguale si verifica un *round robin* semplice.
Ovviamente il multitasking preemptive offre maggiori garanzie nei confronti di vincoli temporali e si presta bene ad essere associato al meccanismo delle interruzioni. Nei multitasking preemptive infatti lo scheduler verifica ad intervalli di tempo regolari, chiamati clock tick, se lo stato dei vari task è cambiato. I clock tick (che sono relativamente frequenti ed in genere sono generati dall'interrupt fornito da un timer hardware) servono anche per aggiornare variabili di stato, contatori e temporizzatori software. Se ad un multitasking preemptive descritto prima si aggiunge anche la caratteristica che tutti i task possono essere eseguiti per un intervallo di tempo massimo prefissato, dopo il quale si ha un cambio automatico di task, si ha un semplice *sistema operativo real-time* (RTOS). La caratteristica principale di un RTOS è che i tempi di esecuzione e risposta sono deterministici o comunque conoscibili con precisione e garantiti, almeno nel caso peggiore.

Sincronizzazione

I vari task possono essere dei programmi completamente indipendenti, tuttavia in molti casi alcuni di essi devono necessariamente collaborare e coordinarsi in qualche modo. Questo può accadere sia quando diversi task sono utilizzati per gestire un unico compito complesso, sia nel caso in cui più task devono utilizzare un'unica risorsa condivisa (una periferica, un'area di memoria, una porta di I/O, etc.). I sistemi operativi mettono a disposizione diversi meccanismi per gestire la coordinazione e la sincronizzazione tra i task, i più comuni sono i *mutex*, i *semafori* e le *code*.
I mutex (contrazione di *mutual exclusion*, cioè mutua esclusione) vengono utilizzati per condividere una risorsa tra due task. Essi funzionano più o meno come un microfono conteso da due presentatori: solo chi ha il microfono in mano può utilizzare l'amplificazione. Nello stesso modo, solo uno dei due task può "prendere" il mutex, l'altro dovrà aspettare che questo venga rilasciato per poterlo prendere a sua volta. Il mutex è quindi una sorta di flag binario, a cui il programmatore può assegnare un significato arbitrario. Ad esempio, se nel sistema è presente una stampante e due task ad un certo punto vogliono utilizzarla, occorrerà fare in modo che uno dei task eviti di mandarle dei caratteri prima che l'altro abbia terminato, altrimenti si otterrà una stampa contenente le due informazioni mischiate! Per fare questo si può creare un mutex, che sarà "preso" dal primo task che dovrà stampare. L'altro task vedrà che il mutex è impegnato e si metterà in attesa fino a quando questo non si libererà. Sarà compito del primo task "rilasciare" il mutex quando avrà finito di stampare.
Se i task da coordinare sono più di due si possono utilizzare i *semafori*. Il loro funzionamento è identico a quello dei mutex, cioè solo uno dei task tra quelli che fanno riferimento allo stesso semaforo potrà aggiudicarsi il "possesso" del semaforo stesso. Il meccanismo è talmente simile a quello dei mutex, che è possibile anche utilizzare questi se il sistema operativo non dispone di semafori (occorrerà solo usare più mutex e gestire il loro stato in maniera coerente). Un altro meccanismo molto utile alla sincronizzazione e comunicazione tra i task è quello delle *code* (dette anche *message queues* o *mail boxes*). Le code vengono utilizzate per passare dei messaggi (o dati) da un task ad un altro. Ad esempio il task che gestisce un display potrebbe ricevere il testo da visualizzare dal task che gestisce la comunicazione seriale via RS232. Il task del display in questo modo potrebbe rimanere in attesa (*waiting*) fino a quando non arriva un messaggio nella coda. Una volta utilizzato il messag-

gio la coda sarebbe di nuovo vuota ed il task andrebbe di nuovo nello stato *waiting*. Le code comunque possono essere utilizzate in modo del tutto arbitrario, è possibile quindi usarle per implementare anche meccanismi più complessi. Se il sistema operativo non supporta le code è sempre possibile rimediare utilizzando un'area di memoria condivisa e qualche mutex o viceversa.

Effetti indesiderati

Il fatto di eseguire diversi task "contemporaneamente", può dar luogo ad una serie di problemi non sempre facilmente intuibili o prevedibili. Alcuni di questi sono simili a quelli visti a proposito delle interruzioni. Ad esempio il problema delle *sezioni critiche* esiste anche in un ambiente multitasking (in particolare se di tipo preemptive), in quanto l'esecuzione di un task può essere interrotta in favore di un altro ed eventuali dati o variabili potrebbero essere scritti in maniera incompleta. Per risolvere questo problema diversi OS mettono a disposizione particolari funzioni per delimitare le sezioni critiche. Anche il problema del salvataggio e del cambio del contesto (*context switch*) da un task ad un altro è molto sentito nel caso degli OS, in quanto esso comporta una perdita di tempo notevole, che va ad incidere alla fine sul "tempo di risposta" dei programmi. Per limitare questo effetto l'unico accorgimento possibile è accorpare più task in uno solo.
Altri effetti indesiderati nascono dall'interazione tra i task (e quindi dall'uso di mutex, semafori e strumenti simili). Il più noto di questi effetti è il *deadlock*, che si verifica quando più task attendono reciprocamente che gli altri compiano una certa azione: alla fine tutti i task rimangono bloccati in una situazione senza uscita.
L'unico modo per uscire da un deadlock spesso è riavviare il sistema! Invece l'unico modo per evitare i deadlock è riflettere bene in fase di progetto sui metodi di sincronizzazione che da utilizzare: alcuni potrebbero portare a *deadlock*, altri sicuramente no, per altri ancora potrebbe essere difficile prevederlo.
Un altro effetto indesiderato nell'uso dei task sincronizzati è *l'inversione di priorità*. Se un task ad alta priorità condivide una risorsa con un task a priorità bassa (tramite un mutex ad esempio), può capitare che il primo resti bloccato per molto tempo in favore del secondo, che impiega molto tempo a completare il suo compito e rilasciare il controllo (il mutex). Anche in questo caso la migliore soluzione è la prevenzione in fase di progetto.

USO DEI SISTEMI OPERATIVI

Per potere utilizzare le funzioni di un sistema operativo è necessario prima di tutto strutturare e scrivere il programma in modo opportuno, secondariamente è necessario "linkare" il codice al sistema operativo stesso. In genere per utilizzare le funzioni del sistema operativo è necessario includere nel codice apposite librerie che permettono di richiamare le funzioni. Il risultato è in alcuni casi un programma indipendente che può essere eseguito dal sistema operativo quando richiesto, in altri casi invece il programma ed il sistema operativo vengono compilati assieme per formare un unico eseguibile che contiene entrambi (questo approccio è più comune nel campo dei sistemi embedded). In quest'ultimo caso si parla di *kernel* o *micro-kernel*. La scrittura del codice è identica nei due casi: occorre soltanto includere gli header del sistema operativo e strutturare il programma in modo opportuno. La struttura di un programma per un OS multitasking è abbastanza diversa da quella di un programma tradizionale. In quest'ultimo caso infatti si ha una funzione `main` che gestisce il flusso di esecuzione principale e molte altre funzioni che vengono richiamate da

questa e il tutto viene eseguito in maniera strettamente sequenziale (se si escludono le interruzioni).
Se il nostro programma intende sfruttare le caratteristiche di multitasking dell'OS, la sua struttura cambia radicalmente: nel `main` verranno creati ed inizializzati i vari task e gli strumenti ausiliari quali mutex, semafori e code. I task verranno poi avviati e saranno quindi eseguiti in concorrenza dallo scheduler. Da questo punto l'esecuzione del programma si divide in più linee di esecuzione, tante quanti sono i task. Per mostrare quanto detto viene considerato di seguito un esempio di sistema operativo reale: FreeRTOS.
FreeRTOS è un semplice ma efficiente kernel real time scritto in ANSI C (con qualche piccola parte in assembler), dotato di tutte le caratteristiche viste prima e disponibile gratuitamente come open source su Internet. FreeRTOS è stato pensato per funzionare su piccoli sistemi a microprocessore o microcontrollore quindi richiede risorse minime per poter essere utilizzato (al contrario di altri OS più noti come uCLinux, eCos o QNX). È disponibile per molti processori, tra cui: ARM7, AVR ATMega, PIC 18x, H8, 8051 e tanti altri. Maggiori informazioni ed i sorgenti possono essere reperiti al sito www.freertos.org.

Creare, eseguire e distruggere i task

Un task non è altro che una normale funzione che contiene un loop infinito (in modo simile a quanto di solito avviene nel `main`). Per creare un task è quindi necessario prima di tutto scrivere la funzione che lo costituisce e assegnarla ad un task utilizzando la funzione `xTaskCreate` di FreeRTOS. Similmente per distruggerlo (quando e se sarà necessario), si potrà utilizzare la funzione `vTaskDelete`:

```
#include "FreeRTOS.h"
#include "task.h"

// Funzione del primo task
void MioTask(void *parametri)
{
  while(1)
  {
    // Codice del task
  }
}

// Funzione del secondo task
void AltroTask(void *parametri)
{
  while(1)
  {
    // Codice del task
  }
}


main()
{
  int parametro=2;
  xTaskHandle MioHandle,
```

```
                AltroHandle;

    // Creazione dei task
    xTaskCreate(MioTask,
                "PRIMO",
                STACK_SIZE,
                &parametro,
                5,
                &MioHandle);

    xTaskCreate(AltroTask,
                "SECONDO",
                STACK_SIZE,
                &parametro,
                2,
                &AltroHandle);

…
    // Avvia esecuzione tasks
    vTaskStartScheduler();
…

    // Distruzione dei tasks.
    vTaskDelete(MioHandle);
    vTaskDelete(AltroHandle);
…
}
```

Per utilizzare FreeRTOS occorre includere i due file header relativi. La funzione `xTaskCreate` prende come parametri il nome della funzione del task (che è a tutti gli effetti un puntatore), il cui prototipo deve essere quello mostrato, una stringa arbitraria che costituisce il nome del task, la dimensione dello stack (cioè il numero di variabili che utilizza la funzione), un puntatore ad una variabile che contiene i parametri da passare alla funzione (che possono essere di qualsiasi tipo, anche strutturato), la priorità del task, che per default è un numero compreso tra 0 e 5 (essendo 5 la massima) ed infine un puntatore ad una variabile in cui verrà restituito *l'handle* al task creato. Un *handle* non è altro che un puntatore, per riferirsi ai task creato. Anche se non è mostrato, la funzione restituisce un valore per comunicare se la creazione è andata a buon fine.
È possibile creare diversi task in questo modo. Quando saranno stati tutti creati, sarà possibile avviare la loro esecuzione, invocando la funzione `vTaskStartScheduler`. A questo punto il controllo del programma passa allo scheduler che gestirà l'esecuzione dei vari task secondo le sue politiche. La funzione `vTaskStartScheduler` non terminerà fino a quando qualcuno dei task eseguiti non invocherà la funzione `vTaskEndScheduler`.
Solo in questo caso la funzione terminerà ed il rimanente codice del `main` riprenderà ad essere eseguito. Si arriverà in questo modo alla funzione `vTaskDelete` che provvederà a distruggere i task.
Un task può essere creato anche durante l'esecuzione del programma, eventualmente da un altro task e può anche auto-distruggersi, secondo le esigenze del programma.

Altre funzioni attinenti ai task

Durante le elaborazioni può succedere che un task abbia esaurito i suoi compiti quindi potrebbe autonomamente scegliere di passare allo stato *waiting*. Per fare questo si possono usare diverse funzioni, alcune fermano il task per un determinato periodo di tempo (misurato in "tick") per poi riportarlo nello stato *ready*:

```
void vTaskDelay(portTickType Ticks);
```

Altre sospendono completamente l'esecuzione del task. L'esecuzione può essere ripristinata ad esempio da un altro task:

```
// Sospende il task
void vTaskSuspend(xTaskHandle pxTaskToSuspend);

// Riprende l'esecuzione
void vTaskResume(xTaskHandle pxTaskToResume);
```

Se un task intende semplicemente cedere il controllo (rimanendo nello stato ready) può usare la macro:

```
taskYIELD
```

Quando all'interno del codice di un task è presente una sezione critica, essa può essere delimitata dalle due macro:

```
taskENTER_CRITICAL
taskEXIT_CRITICAL
```

in questo modo si eviterà si verifichino context switch non voluti all'interno della sezione.

Code

FreeRTOS mette a disposizione le code, che sono internamente utilizzate anche per implementare i mutex ed i semafori. È possibile creare e cancellare code con le seguenti funzioni:

```
xQueueHandle MiaCoda;
MiaCoda=xQueueCreate(10, sizeof(long));
…
vQueueDelete(MiaCoda);
```

In questo modo è stata creata una coda di 10 elementi, ciascuno grande 4 byte (dimensione di una variabile di tipo long). La funzione restituisce un handle alla coda creata o un codice di errore. Per cancellare la coda è sufficiente passare l'handle alla funzione `vQueueDelete`.
Per inviare un messaggio ad una coda si può scrivere:

```
r=xQueueSend(MiaCoda, &datoTX, (portTickType) 10);
```

La funzione prende in ingresso l'handle della coda, il puntatore al dato da inviare ed il tempo (espresso in tick) da aspettare nel caso invece la coda sia piena. In questo tempo il task

passerà allo stato *waiting* e ne uscirà o allo scadere del tempo o quando nella coda si sarà creato spazio. La funzione restituisce un valore che indica l'esito positivo o negativo (coda piena) dell'operazione.
La funzione duale di quella appena vista è la seguente:

```
r=xQueueReceive(MiaCoda, &datoRX, (portTickType) 20);
```

Un task può usare questa funzione per leggere un dato dalla coda o per attenderlo se non è disponibile. Anche in questo caso il task verrà sospeso in attesa dell'esito dell'operazione.

Mutex e semafori

FreeRTOS utilizza internamente il meccanismo delle code per gestire i semafori. In realtà comunque si tratta di semafori binari, quindi di mutex. Per ottenere dei semafori veri e propri è necessario utilizzare più mutex, come già accennato. Un semaforo binario può essere creato con la seguente funzione:

```
xSemaphoreHandle xSemaforo;
vSemaphoreCreateBinary(xSemaforo);
```

Per "prendere" un semaforo si utilizza la seguente funzione:

```
r=xSemaphoreTake(xSemaforo, (portTickType) 30);
```

La funzione specifica, non solo permette di prendere il semaforo (se è disponibile), ma anche di aspettare per un certo periodo di tempo (30 tick) che il semaforo si liberi, prima di concludere che è occupato. Il valore restituito sarà la costante `pdTRUE`, se il semaforo si è liberato ed è stato preso.
Una volta completata l'operazione sulla risorsa condivisa, il task dovrà rilasciare il semaforo utilizzando la seguente funzione:

```
xSemaphoreGive(xSemaforo);
```

CONCLUSIONE

Con le poche e semplici funzioni descritte è possibile scrivere programmi multitasking, che impiegano anche complessi meccanismi di sincronizzazione. Sarà possibile in questo modo non solo rendere più efficienti e modulari i propri programmi, ma anche soddisfare vincoli temporali abbastanza critici, sfruttando le caratteristiche real time di FreeRTOS. Probabilmente apparirà ora più chiaro quali vantaggi può offrire l'uso di un OS. Tuttavia nonostante questo, la scelta di scrivere un programma stand-alone piuttosto che ricorrere ad un OS è lasciata all'esperienza del programmatore: come per gli aspetti visti in precedenza occorre sempre bilanciare bene vantaggi e svantaggi offerti da ciascun approccio.
Per una più completa descrizione di FreeRTOS e delle sue funzioni è consigliabile scaricare la documentazione ed i codici sorgenti dal sito. Tra il materiale scaricabile sono presenti anche dei programmi di esempio, molto utili per iniziare. Anche lo studio degli stessi sorgenti di FreeRTOS può avere un grande valore "didattico", in quanto in essi è possibile ritrovare buona parte degli argomenti trattati in questo libro.

Appendice A

Codici ASCII

CODICE DEC (HEX)	CARATTERE	CODICE DEC (HEX)	CARATTERE	CODICE DEC (HEX)	CARATTERE	CODICE DEC (HEX)	CARATTERE
0 (00)	[NUL]	32 (20)	spazio	64 (40)	@	96 (60)	`
1 (01)	[SOH]	33 (21)	!	65 (41)	A	97 (61)	a
2 (02)	[STX]	34 (22)	"	66 (42)	B	98 (62)	b
3 (03)	[ETX]	35 (23)	#	67 (43)	C	99 (63)	c
4 (04)	[EOT]	36 (24)	$	68 (44)	D	100 (64)	d
5 (05)	[ENQ]	37 (25)	%	69 (45)	E	101 (65)	e
6 (06)	[ACK]	38 (26)	&	70 (46)	F	102 (66)	f
7 (07)	[BEL]	39 (27)	'	71 (47)	G	103 (67)	g
8 (08)	[BS]	40 (28)	(	72 (48)	H	104 (68)	h
9 (09)	[HT]	41 (29)	)	73 (49)	I	105 (69)	i
10 (0A)	[LF]	42 (2A)	*	74 (4A)	J	106 (6A)	j
11 (0B)	[VT]	43 (2B)	+	75 (4B)	K	107 (6B)	k
12 (0C)	[FF]	44 (2C)	,	76 (4C)	L	108 (6C)	l
13 (0D)	[CR]	45 (2D)	-	77 (4D)	M	109 (6D)	m
14 (0E)	[SO]	46 (2E)	.	78 (4E)	N	110 (6E)	n
15 (0F)	[SI]	47 (2F)	/	79 (4F)	O	111 (6F)	o
16 (10)	[DLE]	48 (30)	0	80 (50)	P	112 (70)	p
17 (11)	[DC1]	49 (31)	1	81 (51)	Q	113 (71)	q
18 (12)	[DC2]	50 (32)	2	82 (52)	R	114 (72)	r
19 (13)	[DC3]	51 (33)	3	83 (53)	S	115 (73)	s
20 (14)	[DC4]	52 (34)	4	84 (54)	T	116 (74)	t
21 (15)	[NAK]	53 (35)	5	85 (55)	U	117 (75)	u
22 (16)	[SYN]	54 (36)	6	86 (56)	V	118 (76)	v
23 (17)	[ETB]	55 (37)	7	87 (57)	W	119 (77)	w
24 (18)	[CAN]	56 (38)	8	88 (58)	X	120 (78)	x
25 (19)	[EM]	57 (39)	9	89 (59)	Y	121 (79)	y
26 (1A)	[SUB]	58 (3A)	:	90 (5A)	Z	122 (7A)	z
27 (1B)	[ESC]	59 (3B)	;	91 (5B)	[	123 (7B)	{
28 (1C)	[FS]	60 (3C)	<	92 (5C)	\	124 (7C)	|
29 (1D)	[GS]	61 (3D)	=	93 (5D)	]	125 (7D)	}
30 (1E)	[RS]	62 (3E)	>	94 (5E)	^	126 (7E)	~
31 (1F)	[US]	63 (3F)	?	95 (5F)	_	127 (7F)	[DEL]

Note:

- I codici da 0 a 31 non corrispondono a caratteri stampabili, ma sono “caratteri di controllo”. Alcuni di questi sono utilizzati comunemente: [HT] = tabulazione orizzontale (in C “\t”), [LF] = avanzamento di linea (“\n”), [VT] = tabulazione verticale (“\v”), [FF] = avanzamento pagina (“\f”), [CR] = ritorno carrello (“\r”).
- I codici da 128 a 255 non sono standard e dipendono dal set di caratteri utilizzato o dalla particolare macchina. Di solito questi codici comprendono lettere accentate, simboli vari o caratteri adatti a disegnare semplici finestre, tabelle o riquadri.
- I codici corrispondenti ai simboli numerici (48-57 decimale, 30-39 esadecimale) hanno la proprietà di rappresentare le cifre decimali in BCD se vengono azzerati i 4 bit più significativi del codice. Questo rende più semplice la conversione da testo a BCD/binario e viceversa.

Appendice B

Specificatori di formato (`printf` e `scanf`)

Per specificare il formato di input o output di una variabile si usa il seguente formato:

```
%[flags][width][.prec][F|N|h|l|L]Tipo_dato
```

L'espressione inizia con un %, segue poi una serie di campi opzionali tra cui:

```
[Flags]:
-    giustifica l'output a sinistra invece che a destra;
+    aggiunge il segno + ai numeri positivi.

[Width]:
n    visualizza n cifre, se il numero ha meno cifre vengono
     aggiunti degli spazi vuoti;
0n   visualizza n cifre, aggiungendo eventualmente degli 0 a
     sinistra.

[.Prec]:
.n   stabilisce quante cifre decimali devono essere stampate.

[F|N|h|l|L]:
N = near pointer,
F = far pointer,
h = short int,
l = long,
L = long double.
```

Esempio:

```
printf("- %04Xh -", 254);

- 00FEh -
```

TIPO_DATO	
Numeri:	
%d, %i	Intero decimale con segno
%o	Intero ottale senza segno
%u	Intero decimale senza segno
%x, %X	Intero esadec. senza segno (cifre in minusc. o maiusc.)
%f	Floating point, formato: [-]dddd.dddd
%e, %E	Floating point, formato: [-]d.dddd or e[+/-]ddd (esp. minusc. o maiusc.)
%g, %G	Floating point, formato: come %e oppure %f (esp. minusc. o maiusc.)
Caratteri:	
%c	Caratteri ASCII singoli
%s	Stringa
%	Carattere %
Puntatori:	
%n, %p	Puntatore (il formato dipende dalla macchina)

Appendice C

Complemento a 2 e floating point

Rappresentazione in complemento a 2

La rappresentazione in complemento a 2 è utilizzata nella maggior parte dei sistemi di elaborazione per rappresentare numeri interi con segno. Il vantaggio di questa rappresentazione rispetto ad altre (es. BCD, modulo e segno, etc.) consiste nel fatto che essa permette di eseguire le addizioni e le sottrazioni utilizzando gli stessi circuiti e gli stessi algoritmi. Anche le altre operazioni risultano sostanzialmente uguali al caso di numeri interi senza segno.

Rispetto ad altre rappresentazioni (modulo e segno, complemento ad 1) non ha due zeri (zero positivo e zero negativo) ed è presente un numero negativo in più rispetto ai positivi: l'intervallo rappresentato va quindi da -2^n a 2^n-1.

Nella rappresentazione in complemento a 2 il bit più significativo è legato al segno del numero (vale 0 se positivo, 1 se negativo).

I numeri positivi sono espressi in binario naturale (cioè coincidono con i numeri senza segno). I numeri negativi si ottengono sottraendo il loro valore assoluto alla base: $-N = (2^n - N)$. **Un metodo semplice per convertire un numero da positivo a negativo (e viceversa) consiste nel negare tutti i bit e sommare 1 al numero ottenuto.**

Quello che segue è un esempio di valori a 5 bit in complemento, che permette di apprezzare le considerazioni prima fatte:

```
+15  01111
+14  01110
+13  01101
. . . . .
+3   00011
+2   00010
+1   00001
 0   00000
-1   11111
-2   11110
. . . . .
-14  10010
-15  10000
-16  10000
```

Come già detto il bit più significativo dei numeri negativi deve valere 1. Questo significa che **se si eseguono degli shift a destra e si vuole conservare il segno del numero, occorre introdurre come bit più significativo lo stesso valore che era presente prima dello shift ("shift aritmetico").**

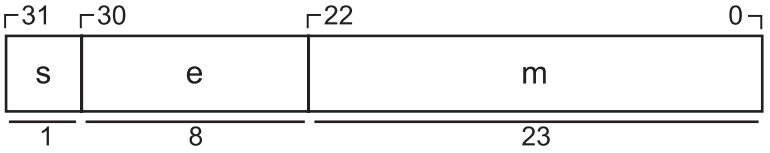
Nello stesso modo se si esegue un casting utilizzando un numero di bit maggiori, tutti i bit aggiunti devono essere posti uguali al bit più significativo del numero originale (questa operazione è nota come *sing extension*).

Rappresentazione floating point

Il formato più utilizzato dai compilatori per rappresentare i tipi a virgola mobile (`float` e `double`, etc...) è quello definito nello standard IEEE 754 del 1985.
Il numero è espresso attraverso una mantissa ed un esponente come nella notazione scientifica, però sono entrambi intesi in binario, ad esempio:

```
11100101 <=> 1.1100101x2^7
```

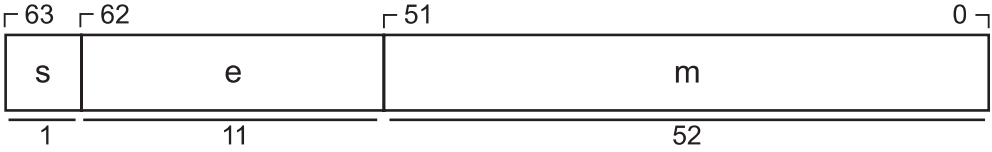
Nello standard a singola precisione si utilizzano 32 bit nel seguente formato:




```
s = 1 bit [31] bit di segno del numero: 0=pos., 1=neg.
e = 8 bit [30-23] per l'esponente
m = 23 bit [22-0] per la mantissa
```

L'esponente è codificato in "eccesso 127", cioè 127=0, 254=127, etc. I valori 0 e 255 non sono utilizzati normalmente. La mantissa viene considerata sempre normalizzata, quindi si dà per scontato che il bit intero sia sempre 1 e per questo non viene rappresentato, ottenendo così in totale 24 bit equivalenti. Lo zero è indicato ponendo tutti i bit di tutti i campi uguali a 0.
Ponendo tutti i bit dell'esponente a 1 si indica il valore non numerico "INF" (infinito) che può essere positivo o negativo. Le operazioni tra INF e numeri finiti danno risultati ben precisi, che sono specificati nello standard.
I numeri che si possono rappresentare vanno da 10^{-44} a 10^{38} sia in positivo che in negativo. La distanza tra un numero e quelli adiacenti non è costante come nella rappresentazione a virgola fissa, infatti dipende dall'esponente ed è più grande per i numeri grandi e più piccola per i numeri piccoli.

Esiste un'estensione dello standard che utilizza 64 bit nel seguente formato:




```
s = 1 bit [63] di segno
e = 11 bit [62-52] per l'esponente codificato in eccesso 1023
m = 52 bit [51-0] per la mantissa.
```

Le convenzioni utilizzate per i vari campi sono le stesse.

Bibliografia

Un riferimento importante sul linguaggio ANSI C:
- *D. Ritchie, B. Kernighan:* **"Il linguaggio C - Corso di programmazione"**, Pearson Education Italia 2004, ISBN 887192200x.

Una discussione più completa su alcuni degli aspetti trattati (strutture dinamiche, complessità computazionale, etc.):
- *S. Cerri, D. Mandrioli. L. Sbattella:* **"Informatica Istituzioni"**, McGraw-Hill 1994, ISBN: 88-386-0703-6.

Approfondimenti su alcuni aspetti relativi alla programmazione dei sistemi embedded in C:
- *Stuart R. Ball:* **"Embedded Microprocessor Systems: Real World Design"**, Butterworth-Heinemann 2000, ISBN 075067234X.
- *Michael Barr:* **"Programming Embedded Systems in C and C++"**, O'Reilly Media 1999, ISBN 1565923545.

La spiegazione e le routine già pronte relative ad alcuni degli argomenti trattati:
- *Jean J. Labrosse:* **"Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C"**, CMP Books 1999, ISBN 0-87930-604-1.

Informazioni sulla rappresentazione dei numeri, l'aritmetica in complemento a 2 e su alcuni codici si trovano in:
- *M. Morris Mano, Charles R. Kime:* **"Reti Logiche"**, Pearson Education Italia 2002, ISBN 887192-142-9.

Una descrizione completa della struttura, dei meccanismi ed il funzionamento dei sistemi operativi:
- *A. Silberschatz, P. Galvin:* **"Sistemi Operativi"**, Addison Wesley 1998. ISBN: 88-7192-064-3.

Il software ed i manuali relativi al compilatore GCC ed al debugger GDB possono essere reperiti su:
- *www.gnu.org*

Il microkernel FreeRTOS e la sua documentazione può essere reperito su:
- *www.freertos.org*

La collana "Conoscere ed usare" comprende i seguenti volumi:

Conoscere ed usare – Display LCD
Questo libro di successo (oltre 2000 copie vendute) rappresenta una delle migliori guide all'utilizzo dei moduli alfanumerici basati sul controller HD44780, moduli grafici con controller KS0108 e non solo. Il testo tratta anche i display LED a sette segmenti e i display LCD passivi. Numerosi gli esempi pratici di impiego dei vari dispositivi: dal contatore a 7 segmenti al termometro LCD fino al pilotaggio dei moduli alfanumerici mediante PICmicro e PC.

Conoscere ed usare – PICmicro™
La lettura di questo libro è consigliata per conoscere a fondo i PICmicro seguendo un percorso estremamente pratico e stimolante. Il testo descrive l'uso di MPLAB® e descrive, in maniera approfondita, tutte le istruzioni assembler e molte delle direttive del compilatore. Al testo è allegato un utilissimo CDROM che, oltre ai sorgenti e gli schemi dei progetti presentati nel testo, contiene moltissimi programmi di utilità e molta documentazione.

Conoscere ed usare – Linguaggio ANSI C
Questo nuovissimo libro descrive le tecniche, gli accorgimenti migliori per sfruttare gli aspetti di "alto e basso livello" del C, entrambi fondamentali quando si vuole sviluppare del firmware per sistemi dotati di risorse limitate. Il testo è particolarmente indicato sia a chi ha già esperienza nella programmazione in assembler di sistemi a microcontrollore ed intende iniziare ad utilizzare il linguaggio C, sia per chi conosce già il C e vuole avvicinarsi alla programmazione dei sistemi embedded.

Conoscere ed usare – BASIC per PIC
Un volume indispensabile sia per chi si avvicina alla programmazione dei PIC utilizzando il linguaggio Basic, sia per chi intende affinare le proprie tecniche di programmazione.
Una guida alla programmazione embedded utilizzando MikroBASIC, uno dei più completi compilatori per PIC dotato di ambiente IDE e moltissime funzioni di libreria. La trattazione vi guiderà dalla semplice accensione di un LED alla gestione di motori in PWM, alla lettura e scrittura di memorie I2C, alla generazione di suoni seguendo un percorso semplice e ricchissimo di esempi pratici.

Conoscere ed usare - CPLD
Un libro dedicato a tutti coloro che per la prima volta si avvicinano al mondo delle Logiche Programmabili ed utilizzabile da quanti, già esperti, desiderano approfondire la conoscenza di questi interessanti dispositivi. Gli argomenti teorici sono presentati attraverso semplici circuiti di esempio il cui codice viene descritto nei dettagli.