Concetti fondamentali sugli Algoritmi

Per molto tempo si pensò che il termine *algoritmo* derivasse da una storpiatura del termine *logaritmo*. L'opinione attualmente diffusa è invece che il termine derivi da *al-Khuwarizmi*, nome derivante a sua volta dal luogo di origine di un matematico arabo, autore di un libro di aritmetica e di uno di algebra: nel libro di aritmetica si parla della cosiddetta numerazione araba (quella attualmente usata) e si descrivono i procedimenti per l'esecuzione delle operazioni dell'aritmetica elementare. Questi procedimenti vennero in seguito chiamati algoritmi e il termine passò ad indicare genericamente qualunque *procedimento di calcolo*.

L'algoritmo esprime le *azioni* da svolgere su determinati *oggetti* al fine di produrre gli *effetti* attesi. Una azione che produce un determinato effetto è chiamata **istruzione** e gli oggetti su cui agiscono le istruzioni possono essere **costanti** (valori che restano sempre uguali nelle diverse esecuzioni dell'algoritmo) e **variabili** (contenitori di valori che variano ad ogni esecuzione dell'algoritmo). Si potrà dire brevemente che un algoritmo è una elaborazione di dati: i dati, cioè l'insieme delle informazioni che devono essere elaborate, sono manipolati, secondo le modalità descritte dalle istruzioni, per produrre altri dati. Ciò porta l'algoritmo ad essere una funzione di trasformazione dei dati di un insieme A (dati di input) in dati di un insieme B (dati di output).

In questi appunti, dato che ci si pone il fine di una introduzione alla programmazione, più che una definizione rigorosa di algoritmo se ne fornirà una definizione intuitiva. In questo senso si può definire l'algoritmo come *".. un insieme di istruzioni che definiscono una sequenza di operazioni mediante le quali si risolvono tutti i problemi di una determinata classe"*.

Per chiarire meglio il concetto di algoritmo è bene fare riferimento ad alcune proprietà che un insieme di istruzioni deve possedere affinché possa chiamarsi algoritmo:

- La **finitezza**. Il numero di istruzioni che fanno parte di un algoritmo è finito. Le operazioni definite in esso vengono eseguite un numero finito di volte.
- Il **determinismo**. Le istruzioni presenti in un algoritmo devono essere definite senza ambiguità. Un algoritmo eseguito più volte e da diversi esecutori, a parità di premesse, deve giungere a medesimi risultati. L'effetto prodotto dalle azioni descritte nell'algoritmo non deve dipendere dall'esecutore o dal tempo.
- La **realizzabilità pratica**. Tutte le azioni descritte devono essere eseguibili con i mezzi di cui si dispone.
- La **generalità**. Proprietà già messa in evidenza nella definizione che si è data: un algoritmo si occupa della risoluzione di famiglie di problemi.

inizio

Tipi di istruzioni

Per quanto osservato nell'ultima proprietà espressa, gli algoritmi operano principalmente su variabili che conterranno i dati sui quali si vuole svolgere una determinata elaborazione. I valori da elaborare devono essere *assegnati* alle variabili prima di effettuare l'elaborazione. Si pensi infatti ad una variabile come ad un contenitore. Le istruzioni operano sui valori contenuti: se questi non ci sono non ci si può attendere alcuna elaborazione. Ogni variabile è identificata da un nome che permette di distinguerla dalle altre.

- L'**istruzione di assegnamento** fa in modo che un determinato valore sia conservato in una variabile. In questo modo si prepara la variabile per l'elaborazione o si conserva nella variabile un valore intermedio prodotto da una elaborazione precedente. Si può assegnare ad una variabile un valore costante come anche il valore risultante da una espressione aritmetica.
- L'**istruzione di input** fa in modo che l'algoritmo riceva dall'esterno un valore da assegnare ad una variabile. Nel caso di algoritmi eseguiti da un elaboratore, questi attende che da una unità di input (per

esempio la tastiera) arrivi una sequenza di caratteri tipicamente terminanti con la pressione del tasto <Invio> . Il dato verrà assegnato alla variabile appositamente predisposta. Praticamente si tratta di una istruzione di assegnamento solo che stavolta il valore da assegnare proviene dall'esterno.
- L'**istruzione di output** fa in modo che l'algoritmo comunichi all'esterno i risultati della propria elaborazione. Nel caso di un elaboratore viene inviato su una unità di output (per esempio il video) il valore contenuto in una determinata variabile.

Esaminiamo adesso due versioni di un algoritmo per il calcolo dell'area di un rettangolo (i nomi delle variabili sono scritti in maiuscolo):

Algoritmo **A**	Algoritmo **B**
`Assegna a BASE il valore 3`	`Ricevi BASE`
`Assegna a ALTEZZA il valore 7`	`Ricevi ALTEZZA`
`Assegna a AREA valore BASE*ALTEZZA`	`Assegna a AREA valore BASE*ALTEZZA`
`Comunica AREA`	`Comunica AREA`

Nell'algoritmo A si assegnano alle variabili `BASE` ed `ALTEZZA` dei valori costanti, l'algoritmo calcola l'area di un rettangolo particolare e se si vuole applicare l'algoritmo ad un diverso rettangolo è necessario modificare le due istruzioni di assegnamento a `BASE` e `ALTEZZA`: l'algoritmo ha perso la sua caratteristica di generalità. Questo esempio non deve portare a concludere che non ha senso parlare di costanti in un algoritmo perché, per esempio, se si fosse trattato di un triangolo il calcolo dell'area avrebbe assunto l'aspetto: `Assegna a AREA valore BASE*ALTEZZA/2`. La costante `2` prescinde dai valori diversi che possono essere assegnati a `BASE` e `ALTEZZA`, essendo parte della formula del calcolo dell'area di un triangolo *qualsiasi*.

Nell'algoritmo B i valori da assegnare a `BASE` e `ALTEZZA` provengono da una unità di input. L'algoritmo ad ogni esecuzione si fermerà in attesa di tali valori e il calcolo verrà eseguito su tali valori: l'elaborazione è sempre la stessa (l'area di un rettangolo si calcola sempre allo stesso modo) ma i dati saranno di volta in volta diversi (i rettangoli hanno dimensioni diverse).

inizio

Le strutture di controllo

Gli algoritmi, a causa della loro generalità, lavorano utilizzando variabili. Non si conoscono, al momento della stesura dell'algoritmo stesso, i valori che possono assumere le variabili. Ciò se permette di scrivere algoritmi generali può comportare problemi per alcune istruzioni: si pensi al problema apparentemente banale del calcolo del quoziente di due numeri:

```
Ricevi DIVIDENDO

Ricevi DIVISORE

Assegna a QUOZIENTE valore DIVIDENDO/DIVISORE

Comunica QUOZIENTE
```

Il quoziente può essere calcolato se `DIVISORE` contiene un valore diverso da `0`, evento questo non noto in questo momento dipendendo tale valore dal numero proveniente da input. Inoltre è chiaro che, in questa eventualità, sarebbe priva di senso anche l'istruzione di output del valore di `QUOZIENTE` non essendoci nella variabile alcun valore.

È necessario introdurre, oltre alle istruzioni, degli strumenti che permettano di controllare l'esecuzione dell'algoritmo: le strutture di controllo. La **programmazione strutturata** (disciplina nata alla fine degli anni

60 per stabilire le regole per la scrittura di buoni algoritmi) impone l'uso di tre sole regole di composizione degli algoritmi:

la sequenza

È l'unica struttura di composizione che si è utilizzata finora. In poche parole questa struttura permette di specificare l'ordine con cui le istruzioni si susseguono: ogni istruzione produce un risultato perché inserita in un contesto che è quello determinato dalle istruzioni che la precedono. Nell'esempio di prima il calcolo di `QUOZIENTE`, per poter contenere il valore atteso, deve essere eseguito dopo gli input.

la selezione

Questa struttura permette di scegliere tra due alternative la sequenza di esecuzione. È la struttura che ci permette, per esempio, di risolvere in modo completo il problema del calcolo del quoziente fra due numeri:

```
Ricevi DIVIDENDO

Ricevi DIVISORE

Se DIVISORE ≠ 0

Assegna a QUOZIENTE valore DIVIDENDO/DIVISORE

Comunica QUOZIENTE

Altrimenti

Comunica 'Operazione senza senso'

Fine-se
```

La condizione espressa nella struttura `Se` permette di scegliere, in relazione al valore di verità o falsità, quale elaborazione svolgere. La sequenza contenuta nella parte `Altrimenti` potrebbe mancare se si volesse soltanto un risultato laddove possibile: in tale caso se la condizione `DIVISORE ≠ 0` risultasse non verificata, non si effettuerebbe alcuna elaborazione.

l'iterazione

La struttura iterativa permette di ripetere più volte la stessa sequenza di istruzioni finché non si verifica una determinata condizione. Si voglia, a titolo di esempio, *scrivere un algoritmo che calcoli e visualizzi i quadrati di una serie di numeri positivi*. Si tratta in altri termini di effettuare la stessa elaborazione (calcolo e visualizzazione del quadrato di un numero) effettuata su numeri diversi (quelli che arriveranno dall'input):

```
Ricevi NUMERO

Mentre NUMERO > 0

Assegna a QUADRATO valore NUMERO*NUMERO

Comunica QUADRATO

Ricevi NUMERO

Fine-mentre
```

Dentro la struttura iterativa (la parte compresa fra le parole `Mentre` e `Fine-mentre`) sono specificate le istruzioni per il calcolo del quadrato di un numero: l'iterazione permette di ripetere tale calcolo per tutti i numeri che verranno acquisiti tramite l'istruzione di input inserita nell'iterazione stessa. La condizione

`NUMERO>0` viene chiamata **condizione di controllo del ciclo** e specifica quando deve terminare l'elaborazione (il valore introdotto da input è non positivo): si ricorda che l'algoritmo deve essere finito e non si può iterare all'infinito. Il primo input fuori ciclo ha lo scopo di permettere l'impostazione della condizione di controllo sul ciclo stesso e stabilire, quindi, quando terminare le iterazioni.

In generale si può dire che la struttura di una elaborazione ciclica, controllata dal verificarsi di una condizione, assume il seguente aspetto:

Considera primo elemento

Mentre elementi non finiti

Elabora elemento

Considera prossimo elemento

Fine mentre

Le strutture di controllo devono essere pensate come schemi di composizione: una sequenza può contenere una iterazione che, a sua volta, contiene una selezione ecc.. Rappresentano in pratica i mattoncini elementari di una scatola di montaggio le cui diverse combinazioni permettono la costruzione di architetture di varia complessità.

inizio

Accumulatori e contatori

L'elaborazione ciclica è spesso utilizzata per l'aggiornamento di totalizzatori o contatori. Per chiarire meglio il concetto di totalizzatore, si pensi alle azioni eseguite dal cassiere di un supermercato quando si presenta un cliente con il proprio carrello pieno di merce. Il cassiere effettua una elaborazione ciclica sulla merce acquistata: ogni oggetto viene esaminato per acquisirne il prezzo. Lo scopo della elaborazione è quello di cumulare i prezzi dei prodotti acquistati per stabilire il totale che il cliente dovrà corrispondere.

Dal punto di vista informatico si tratta di utilizzare una variabile (nell'esempio potrebbe essere rappresentata dal totalizzatore di cassa) che viene aggiornata per ogni prezzo acquisito: ogni nuovo prezzo acquisito non deve sostituire il precedente ma aggiungersi ai prezzi già acquisiti precedentemente. Tale variabile:

1. dovrà essere azzerata quando si passa ad un nuovo cliente (ogni cliente dovrà corrispondere solamente il prezzo dei prodotti che lui acquista)
2. si aggiornerà per ogni prodotto esaminato (ogni nuovo prezzo acquisito verrà cumulato ai precedenti)
3. finito l'esame dei prodotti acquistati la variabile conterrà il valore totale da corrispondere.

La variabile di cui si parla nell'esempio è quella che, nel linguaggio della programmazione, viene definita un **totalizzatore** o **accumulatore**: cioè una variabile nella quale ogni nuovo valore non sostituisce ma si accumula a quelli già presenti in precedenza. Se la variabile si aggiorna sempre di una quantità costante (per esempio viene sempre aggiunta l'unità) viene chiamata **contatore**.

In generale si può dire che l'uso di un totalizzatore prevede i seguenti passi:

Inizializzazione totalizzatore

Inizio ciclo aggiornamento totalizzatore

…

Aggiornamento totalizzatore

Fine ciclo

L'inizializzazione serve sia a dare senso all'istruzione di aggiornamento (cosa significherebbe la frase: *aggiorna il valore esistente con il nuovo valore* se non ci fosse un valore esistente ?), sia a fare in modo che l'accumulatore stesso contenga un valore coerente con l'elaborazione da svolgere (nell'esempio di prima il nuovo cliente non può pagare prodotti acquistati dal cliente precedente: il totalizzatore deve essere azzerato, prima di cominciare l'elaborazione, affinché contenga un valore che rispecchi esattamente tutto ciò che è stato acquistato dal cliente esaminato).

L'aggiornamento viene effettuato all'interno di un ciclo. Se infatti si riflette sulla definizione stessa di totalizzatore, è facile prendere atto che avrebbe poco significato fuori da un ciclo: come si può cumulare valori se non si hanno una serie di valori ?

Quando i valori da esaminare terminano, il totalizzatore conterrà il valore cercato. Nell'esempio di prima tutto ciò si tradurrebbe: finito l'esame dei prodotti acquistati, si potrà presentare al cliente il totale da corrispondere.

Si voglia, a titolo di esempio di utilizzo di un accumulatore, risolvere il seguente problema: *data una sequenza di numeri positivi, se ne vuole calcolare la somma*.

```
Inizializza SOMMA con valore 0

Ricevi NUMERO

Mentre NUMERO > 0

Aggiorna SOMMA sommando NUMERO

Ricevi NUMERO

Fine-mentre

Comunica SOMMA
```

Il totalizzatore `SOMMA` è inizializzato, prima del ciclo, al valore nullo poiché deve rispecchiare la somma dei numeri introdotti da input e, quindi, non essendo ancora stata effettuata alcuna elaborazione su alcun numero tale situazione viene espressa assegnando il valore neutro della somma.

L'output di `SOMMA` alla fine del ciclo indica il fatto che si può utilizzare (in questo caso per la conoscenza del valore contenuto) il totalizzatore quando il suo contenuto è coerente con il motivo della sua esistenza: `SOMMA` deve accumulare tutti i valori e ciò avverrà quando tutti i numeri da considerare saranno stati elaborati, cioè in uscita dal ciclo.

I linguaggi di programmazione

I linguaggi di programmazione permettono di scrivere algoritmi interpretabili da un sistema di elaborazione. Un algoritmo scritto in un linguaggio di programmazione viene chiamato **programma** e il processo di scrittura del programma, a partire dall'algoritmo, viene chiamato **codifica**. I linguaggi di programmazione sono costituiti da un alfabeto e da un insieme di regole che devono essere rispettate per scrivere programmi sintatticamente corretti.

Il linguaggio macchina costituito da zero ed uno è l'unico che pilota direttamente le unità fisiche dell'elaboratore in quanto è l'unico comprensibile dall'elaboratore stesso. È però estremamente complicato scrivere programmi in tale linguaggio naturale per la macchina ma completamente *innaturale* per l'uomo. Per poter permettere un dialogo più semplice con la macchina sono nati i linguaggi di programmazione.

Il più *vecchio* linguaggio di programmazione è il linguaggio assembly. Il linguaggio assembly è una rappresentazione simbolica del linguaggio macchina. La scrittura di programmi è enormemente semplificata rispetto a quest'ultimo. Per essere eseguito dall'elaboratore un programma in linguaggio assembly deve essere tradotto in linguaggio macchina; tale lavoro è a carico di un programma detto *assemblatore.* Questi due tipi di linguaggi, detti anche linguaggi di *basso livello* sono propri di ogni macchina.

I linguaggi di *alto livello* sono più vicini al linguaggio naturale, sono orientati ai problemi piuttosto che all'architettura della macchina. Non fanno riferimento ai registri fisicamente presenti sulla macchina ma a variabili. Per essere eseguiti devono essere tradotti in linguaggio macchina, e tale traduzione viene svolta da un programma detto **compilatore**.

I linguaggi di alto livello sono in larga misura indipendenti dalla macchina, possono essere eseguiti su qualsiasi elaboratore a patto che esista il corrispondente compilatore che ne permetta la traduzione.

I linguaggi di alto livello si caratterizzano per essere orientati a specifiche aree applicative. Questi linguaggi vengono anche detti della terza generazione.

Per ultimi in ordine di tempo sono arrivati i linguaggi della quarta generazione, ancora più spiccatamente rivolti a specifiche aree applicative e, nell'ambito del loro orientamento, utilizzabili in modo intuivo dall'utente non esperto. Il più famoso di questi è SQL (Structured Query Language), che opera su basi dati relazionali. I linguaggi di IV generazione sono detti *non procedurali* poiché l'utente specifica la funzione che vuole svolgere senza entrare nel dettaglio di come verrà effettivamente svolta.

inizio

Il linguaggio C

Nel 1972, presso i Bell Laboratories, Dennis Ritchie progettava e realizzava la prima versione del linguaggio C. Ritchie aveva ripreso e sviluppato molti dei principi e dei costrutti sintattici del linguaggio BCPL, sviluppato da Martin Richards, e del linguaggio B, sviluppato da Ken Thompson, l'autore del sistema operativo Unix. Successivamente gli stessi Ritchie e Thompson riscrissero in C il codice di Unix.

Il C si distingueva dai suoi predecessori per il fatto di implementare una vasta gamma di tipi di dati (carattere, interi, numeri in virgola mobile, strutture) non originariamente previsti dagli altri due linguaggi. Da allora ad oggi il C ha subito trasformazioni: la sua sintassi è stata affinata, soprattutto in conseguenza della estensione *object-oriented (C++)*. Nel 1983, l'Istituto Nazionale Americano per gli Standard (ANSI) ha costituito un comitato per *una definizione del linguaggio C non ambigua e non dipendente dalla macchina*: il risultato è lo standard ANSI per il C. A questo standard si farà riferimento in questi appunti.

inizio

Struttura di un programma

Iniziamo esaminando il programma del seguente listato.

```
#include<stdio.h>

main()

{

    printf("abc");

    printf("def");

    printf("ghi");

    printf("lmn");

    printf("opqrs");

    printf("tuvz");

}
```

Eseguendolo verrà visualizzata la stringa delle lettere dell'alfabeto italiano:

```
abcdefghilmnopqrstuvz
```

Dalla parola `main`, seguita da parentesi tonda aperta e chiusa, inizia l'esecuzione del programma. Il corpo del programma, che comincia dalla parentesi graffa aperta e finisce alla parentesi graffa chiusa, è composto da una serie di istruzioni `printf` che verranno eseguite sequenzialmente.

L'istruzione `printf` permette la stampa su video di ciò che è racchiuso tra parentesi tonde e doppi apici. Per esempio

```
printf("abc");
```

visualizza

```
abc
```

Ogni istruzione deve terminare con un carattere di punto e virgola.

Per poter utilizzare `printf`, come le altre funzioni di entrata/uscita, si deve inserire all'inizio del testo la linea

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

che avverte il compilatore di includere i riferimenti alla libreria standard di input/output (`stdio` sta per *standard input/output).*

Il C distingue tra lettere maiuscole e minuscole; dunque occorre fare attenzione, se si scrive MAIN() o Main() non si fa riferimento a `main()`.

La struttura del programma C che abbiamo usato nell'esempio è:

```
inclusione_librerie

main()

{

   istruzione1;

   istruzione2;

   istruzione3;

   .....

   istruzioneN;

}
```

Il punto e virgola conclude l'istruzione.

Se si desidera che l'uscita di ogni istruzione `printf` venga prodotta su una linea separata, si deve inserire **`\n`** al termine di ogni stringa e prima della chiusura dei doppi apici, come nel seguente Listato.

```
#include <stdio.h>

main()

{

    printf("abc\n");

    printf("def\n");

    printf("ghi\n");

    printf("lmn\n");

    printf("opqrs\n");

    printf("tuvz\n");

}
```

Eseguendo il programma si otterrà la visualizzazione delle seguenti stringhe di caratteri

```
abc

def

ghi

lmn

opqrs

tuvz
```

In questo caso la prima stringa (`abc`) viene stampata su video a partire dalla posizione attuale del cursore. Se si vuole cominciare la stampa delle stringhe da una nuova riga basta inserire **`\n`** anche all'inizio come in:

```
printf("\nabc\n");
```

Qui *prima* si passa ad una nuova riga, *poi* si stampa la stringa specificata e *quindi* si posiziona il cursore in una nuova riga.

In generale è bene tenere presente che ogni `printf` stampa a partire dalla posizione in cui si trovava il cursore (in generale a destra dell'ultima stampa). Per poter modificare tale comportamento è necessario inserire gli opportuni caratteri di controllo. In effetti la sequenza **`\n`** corrisponde ad un solo carattere, quello di nuova linea *(newline)*.

Nella `printf` possono inserirsi altri caratteri di controllo di cui si forniscono di seguito quelli che possono avere utilizzo più frequente:

- **`\n`** porta il cursore all'inizio della riga successiva
- **`\t`** porta il cursore al prossimo fermo di tabulazione (ogni fermo di tabulazione è fissato ad 8 caratteri)
- **`\a`** (*alert*) fa emettere un suono dallo speaker
- **`\'`** stampa un apice
- **`\"`** stampa le virgolette

inizio

Variabili e assegnamenti

Supponiamo di voler calcolare l'area di un rettangolo i cui lati hanno valori interi. Entrano in gioco due variabili, la base e l'altezza, il cui prodotto è ancora un valore intero, l'area appunto.

```
#include <stdio.h>

/* Calcolo area rettangolo */

main()

{

    /* dichiarazione delle veriabili */

    int base;

    int altezza;

    int area;



    /* parte elaborativa */

    base = 3;

    altezza = 7;

    area = base*altezza;



    /* output dei risultati */

    printf("%d",area);
```

```
}/* fine programma */
```

Per rendere evidente la funzione espletata dal programma si è inserito un commento:

```
/* Calcolo area rettangolo */
```

I commenti possono estendersi su più linee e apparire in qualsiasi parte del programma, devono essere preceduti da `/*` e seguiti da `*/`, tutto ciò che appare nelle zone così racchiuse non viene preso in considerazione dal compilatore e non ha alcuna influenza sul funzionamento del programma, è però importantissimo per chi legge il programma: è infatti nelle righe di commento che viene specificato il senso delle istruzioni che seguiranno. Cosa, questa, non immediatamente comprensibile se si leggono semplicemente le istruzioni del linguaggio.

Subito dopo `main()` sono presenti le dichiarazioni delle variabili intere necessarie:

```
int base; int altezza; int area;
```

La parola chiave `int` specifica che l'identificatore che lo segue si riferisce ad una variabile di tipo intero; dunque `base`, `altezza` e `area` sono variabili di questo tipo.

Anche le dichiarazioni così come le altre istruzioni devono terminare con un punto e virgola. Nel nostro esempio alla dichiarazione del tipo della variabile corrisponde anche la sua definizione che fa sì che le venga riservato uno spazio in memoria centrale.

Il nome di una variabile la identifica, il suo tipo ne definisce la dimensione e l'insieme delle operazioni che vi si possono effettuare. La dimensione può variare rispetto all'implementazione; molte versioni del C, come quelle sotto i sistemi operativi MS-DOS e Unix, riservano per gli `int` uno spazio di due byte, il che permette di poter lavorare su interi che vanno da -32768 a +32767. Tra le operazioni permesse fra `int` vi sono: la somma (+), la sottrazione (−), il prodotto (*) e la divisione (/).

Effettuata la dichiarazione, la variabile può essere utilizzata. L'istruzione

```
base = 3;
```

assegna alla variabile base il valore 3; cioè inserisce nello spazio di memoria riservato a tale variabile il valore indicato. Effetto analogo avrà `altezza=7`. L'assegnamento è dunque realizzato mediante l'operatore = .

Nel linguaggio C è possibile assegnare lo **stesso valore** a più variabili contemporaneamente. Per esempio se le dimensioni riguardavano un quadrato, si sarebbe potuto scrivere:

```
base = altezza = 5;
```

In questo caso prima verrebbe assegnato il valore `5` alla variabile `altezza` e quindi, il risultato dell'assegnazione (cioè `5`), viene assegnato alla variabile `base`.

L'istruzione:

```
area = base * altezza;
```

assegna alla variabile `area` il prodotto dei valori di `base` e `altezza` .

L'operatore asterisco effettua l'operazione di prodotto tra la variabile che lo precede e quella che lo segue, è dunque un operatore binario.

L'ultima istruzione

```
printf("%d",area);
```

visualizza `21`, il valore della variabile `area`. Tra i doppi apici, il simbolo di percentuale `%` specifica che il carattere che lo segue definisce il **formato di stampa** della variabile `area`; `d` (decimale) indica che si tratta di un intero del quale si desidera la visualizzazione nel sistema decimale.

Le dichiarazioni delle variabili dello stesso tipo possono essere scritte in sequenza separate da una virgola:

```
int base,altezza,area;
```

Dopo la dichiarazione di tipo sono specificati gli identificatori di variabile, che possono essere in numero qualsiasi, separati da virgola e chiusi da un punto e virgola. In generale quindi la dichiarazione di variabili ha la seguente forma:

```
tipo lista_di identificatori;
```

Esistono inoltre delle regole da rispettare nella costruzione degli identificatori: devono iniziare con una lettera o con un carattere di sottolineatura _ e possono contenere lettere, cifre e il carattere di sottolinuatura (under-score) _. Per quanto riguarda la lunghezza occorre tenere presente che soltanto i primi trentadue caratteri sono significativi, anche se nelle versioni del C meno recenti questo limite scende a otto caratteri. Sarebbe comunque opportuno non iniziare il nome della variabile con il carattere di sottolineatura ed è bene tenere presente che le lettere accentate, permesse dalla lingua italiana, non sono considerate lettere ma segni grafici e le lettere maiuscole sono considerate diverse dalle rispettive minuscole.

Oltre a rispettare le regole precedentemente enunciate, un identificatore **non può essere una parola chiave del linguaggio**, **né può essere uguale ad un nome di funzione libreria o scritta dal programmatore**.

Allo scopo di rendere più chiaro l'effetto ottenuto dal programma dell'esempio precedente, si possono visualizzare i valori delle variabili base e altezza.

```
printf("%d ",base);

printf("%d ",altezza);

printf("%d",area);
```

Nelle prime due `printf` si è inserito all'interno dei doppi apici, di seguito all'indicazione del formato di stampa `%d`, uno spazio, in modo che venga riportato, in fase di visualizzazione, dopo il valore della base e dell'altezza, così da ottenere

```
3 7 21
```

e non

```
3721
```

È opportuno, per motivi di chiarezza, far precedere la visualizzazione dei valori da una descrizione. In questo caso è sufficiente inserirla tra doppi apici.

```
printf("Base: %d ",base);

printf("Altezza: %d ",altezza);

printf("Area: %d",area);
```

Quello che si ottiene in esecuzlone è

```
Base: 3 Altezza: 7 Area: 21
```

Per ottenere una distanza maggiore tra il valore di base e altezza e le seguenti descrizioni si possono aggiungere ulteriori spazi

```
printf("Base: %d",base);

printf("\tAltezza: %d",altezza);

printf("\tArea: %d",area);
```

così da avere

```
Base: 3    Altezza: 7    Area: 21
```

Per far in modo che ad ogni visualizzazione corrisponda un salto riga si deve inserire **\n** prima della chiusura dei doppi apici:

```
printf("Base: %d\n",base);

printf("Altezza: %d\n",altezza);

printf("Area: %d\n",area);
```

Si possono effettuare più output con la stessa istruzione. Per esempio:

```
printf("Base: %d Altezza: %d Area: %d",base,altezza,area);
```

visualizzerebbe:

```
Base: 3 Altezza: 7 Area: 21
```

In questo caso figura una lista di variabili da mandare in output. In sede di esecuzione ogni simbolo di formato si riferisce ad una variabile (il primo `%d` si riferisce a `base`, il secondo ad `altezza` ecc…). Ovviamente si sarebbero potuti pure inserire diversi caratteri di formato per formattare, a seconda delle preferenze, l'output (per es **\n** per andare a capo e **\t** per distanziare).

Osserviamo nel listato il programma C modificato seguendo alcune delle caratteristiche introdotte.

```
#include <stdio.h>

/* Calcolo area rettangolo */

main()

{

    /* dichiarazione variabili */

    int base,altezza,area;



    /* parte elaborativa */

    base = 3;

    altezza = 7;

    area = base*altezza;
```

```
    /* output dei risultati */

    printf("Base: %d Altezza: %d\n Area: %d",base,altezza,area);


}/* fine programma */
```

L'esecuzione del programma avrà il seguente effetto:

```
Base: 3 Altezza: 7

Area: 21
```

Mentre `int` è una parola chiave del C e fa parte integrante del linguaggio, `base`, `altezza` e `area` sono identificatori di variabili scelti a nostra discrezione. Lo stesso effetto avremmo ottenuto utilizzando al loro posto altri nomi generici quali `x`, `y` e `z` solo che il programma sarebbe risultato meno comprensibile.

La forma grafica data al programma è del tutto opzionale; una volta rispettata la sequenzialità e la sintassi, la scrittura del codice è libera. In particolare più istruzioni possono essere scritte sulla stessa linea. E indubbio che il programma risulterà notevolmente meno leggibile del precedente.
E' quindi consigliato scrivere le istruzioni su righe diverse e separare i blocchi di istruzioni in funzione del loro significato utilizzando i commenti per rendere il codice facilmente leggibile.

Lo stile grafico facilita enormemente il riconoscimento dei vari pezzi di programma e consente una diminuzione di tempo nelle modifiche, negli ampliamenti e nella correzione degli errori. In generale è inoltre bene dare alle variabili dei nomi significativi, in modo che, quando si debba intervenire a distanza di tempo sullo stesso programma, si possa facilmente ricostruire l'uso che si è fatto di una certa variabile.

inizio

Costanti

Nel programma visto precedentemente i valori di `base` e `altezza` sono costanti, dato che non variano durante l'esecuzione del programma. Evidentemente avremmo potuto scrivere direttamente

```
area = 3 * 7;
```

Quando un certo valore viene utilizzato in modo ricorrente è opportuno rimpiazzarlo con un nome simbolico; per farlo dobbiamo definire, all'inizio del programma, mediante l'istruzione `define` un identificatore di costante in corrispondenza del valore desiderato.
Queste sono appunto dette *costanti simboliche*:

```
#define BASE 3
```

Grazie a questa istruzione potremo utilizzare, all'interno del programma, `BASE` al posto del valore intero `3`.

La stessa definizione di costante implica che il suo valore non può essere modificato: `BASE` può essere utilizzata in un'espressione a patto che su di essa non venga mai effettuato un assegnamento.

Vediamo nel Listato come viene modificato il programma del paragrafo precedente con l'utilizzazione delle costanti.

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

```
/* dichiarazione costanti simboliche */

#define BASE 3

#define ALTEZZA 7


/* Calcolo area rettangolo */

main()

{

    /* dichiarazione variabili */

    int area;


    /* parte elaborativa */

    area = BASE * ALTEZZA;


    /* output dei risultati */

    printf("Base: %d\n",BASE);

    printf("Altezza: %d\n",ALTEZZA);

    printf("Area: %d\n",area);


}/* fine programma */
```

Il nome di una costante può essere qualsiasi identificatore valido in C, comunque abbiamo scelto di utilizzare esclusivamente caratteri maiuscoli per le costanti e caratteri minuscoli per le variabili per distinguere chiaramente le une dalle altre. Le costanti `BASE` e `ALTEZZA` vengono considerate di tipo intero in quanto il loro valore è costituito da numeri senza componente frazionaria.

Invece di utilizzare direttamente i valori, è consigliabile far uso degli identificatori di costante che sono descrittivi e quindi migliorano la leggibilità dei programmi. Inoltre, se ci si rende conto che un certo valore utilizzato più volte deve essere cambiato, nella prima ipotesi, è necessario ricercalo con attenzione all'interno del testo e modificarlo dov'è il caso, nella seconda ipotesi è sufficiente intervenire sulla sua definizione. Per esempio, per fare in modo che il programma precedente calcoli l'area del rettangolo con base `102` e altezza `34`, è sufficiente modificare le linee dov'è presente l'istruzione `define`.

```
#define BASE 102

#define ALTEZZA 34
```

In sintesi, l'uso delle costanti migliora due parametri classici di valutazione dei programmi: **flessibilità** e **possibilità di manutenzione**.

La `define` è in realtà una macroistruzione (brevemente, *macro)* del precompilatore C che offre altre possibilità oltre a quella di definire delle costanti.

inizio

Incrementare una variabile

Ogni nuova assegnazione ad una variabile, distrugge un valore precedentemente contenuto nella variabile stessa. Per cui, nel successivo esempio:

```
...

voto = 3;

...

voto = 6;
```

la variabile `voto` varrà `6`. E ciò qualunque sia stato il valore precedente.

Per **aggiungere** uno al valore contenuto in una variabile si deve assegnare alla variabile il **valore precedente** più uno. Ad esempio, questa istruzione aggiunge uno al contenuto della variabile n:

```
n = n+1;
```

Difatti il C calcola per prima cosa il valore dell'espressione posta a destra del segno di uguale, cioè `n+1`. Se ad esempio `n` vale `5`, `n+1` vale `6`. Questo valore viene poi assegnato alla variabile indicata a sinistra, cioè alla stessa `n`. Quindi `n`, che prima valeva `5`, dopo l'esecuzione dell'istruzione vale `6`.

Questo si chiama *incrementare* una variabile, cioè appunto aggiungere un nuovo valore al valore precedente (nel caso esaminato si aggiunge `1`). L'operazione opposta (togliere, per esempio, `1` al valore della variabile) è chiamata *decrementare* la variabile. Ovviamente si può usare lo stesso sistema per compiere qualunque operazione sul contenuto della variabile:

```
area = area*2; /* raddoppia l'area */

segmento = segmento/k; /* divide il segmento per k */
```

Il linguaggio C dispone di un operatore speciale per incrementare una variabile di una unità. Scrivere:

```
contakm++;
```

equivale a scrivere:

```
contakm = contakm+1;
```

Cioè ad incrementare di una unità il valore della variabile `contakm`. L'operatore `++` è l'operatore di **autoincremento**. L'operatore reciproco `--` (due simboli meno) *decrementa* di una unità il valore di una variabile:

```
altezza--; /* riduce l'altezza di 1 */
```

L'operatore `--` è quindi l'operatore di **autodecremento**.

Si sono viste, quindi, due tecniche per aggiungere uno al valore contenuto in una variabile:

```
fogli = fogli+1;
```

```
fogli++;
```

Esiste anche una terza tecnica:

```
fogli += 1;
```

La combinazione += è un esempio di **operatore di assegnazione**. L'istruzione:

```
km += 1;
```

si può leggere: aggiungi uno al valore corrente della variabile `km`. L'operatore += non è limitato ad aggiungere `1` ma *somma* il valore alla sua destra alla variabile alla sua sinistra. Ad esempio:

```
km += 37;          k1 += k2;          a += (b/2);
```

equivalgono rispettivamente a:

```
km = km+37;        k1 = k1+k2;        a = a+(b/2);
```

L'operatore += non è l'unico operatore di assegnazione, si possono usare *tutti gli operatori aritmetici*:

```
km -= 6; /* toglie 6 ai km percorsi */

lato *= 2; /* moltiplica il lato per 2 */

volume /= 3; /* divide il volume per 3 */

...
```

Nel seguito di questi appunti si converrà di utilizzare:

- gli operatori di autoincremento e di autodecremento tutte le volte che una variabile dovrà essere aggiornata con l'unità
- il doppio operatore (es. +=, −= ecc…) tutte le volte che si parlerà di aggiornamento generico di una variabile (per es. negli accumulatori)

- l'operatore di assegnamento generico (cioè =) in tutti gli altri casi.

inizio

Pre e post-incremento

Per aggiungere uno alla variabile `z` si può scrivere in due modi:

```
z++;                                    ++z;
```

cioè mettere l'operatore ++ prima o dopo del nome della variabile.

In generale, le due forme sono equivalenti. La differenza importa solo quando si scrive una *espressione* che contiene `z++` o `++z`.

Scrivendo `z++`, il valore di `z` viene *prima usato poi incrementato*:

```
int x,z; /* due variabili intere */

z = 4; /* z vale 4 */

x = z++; /* anche x vale 4 ma z vale 5 */
```

Difatti, *prima* il valore di `z` (`4`) è stato assegnato ad `x`, *poi* il valore di `z` è stato incrementato a `5`.

Scrivendo `++z`, il valore di z viene *prima incrementato e poi usato*:

```
int x,z; /* due variabili intere */

z = 4; /* z vale 4 */

x = ++z; /* ora x vale 5 come z */
```

Difatti, *prima* il valore di `z` (`4`) è stato incrementato a `5`, *poi* il nuovo valore di `z` (`5`) è stato assegnato ad `x`.

inizio

Immissione ed emissione di dati

Il programma scritto non calcola l'area di un qualsiasi rettangolo ma soltanto di quello che ha per base `3` e per altezza `7`, supponiamo centimetri.

Per esempio, per trovare l'area di un rettangolo di base `57` e altezza `20` si deve intervenire sul programma stesso, scrivendo

```
#define BASE 57

#define ALTEZZA 20
```

Oppure nel caso della versione del programma dove non si sono utilizzate le costanti ma le variabili, dovremmo scrivere

```
base = 57;

altezza = 20:
```

Dopo aver effettuato nuovamente la compilazione, la successiva esecuzione restituirà `1140`, cioè l'area del rettangolo in centimetri quadri.

Per rendere il programma più generale, si deve permettere a chi lo sta utilizzando di immettere i valori della base e dell'altezza; in questo modo l'algoritmo calcolerà l'area di un qualsiasi rettangolo.

```
scanf("%d",&base)
```

L'esecuzione di questa istruzione fa sì che il sistema attenda l'immissione di un dato da parte dell'utente.

Analogamente a quello che accadeva in `printf`, `%d` indica che si tratta di un valore intero in formato decimale, tale valore verrà poi assegnato alla variabile `base` .

Si presti attenzione al fatto che in una istruzione `scanf` il simbolo `&` (e commerciale, *ampersand)* deve precedere immediatamente il nome della variabile; `&base` sta ad indicare l'indirizzo di memoria in cui si trova la variabile `base`. È infatti noto che le locazioni di memoria in cui sono conservati i dati, sono distinguibili una dall'altra mediante un indirizzo. Il nome della variabile è una etichetta che viene assegnata, per nostra comodità, ad una o più locazioni di memoria consecutive. L'istruzione `scanf("%d",&base);` può allora essere così interpretata: *leggi un dato intero e collocalo nella posizione di memoria il cui indirizzo è &base* (praticamente l'indirizzo corrispondente alla variabile `base`).

Durante l'esecuzione di un programma può essere richiesta all'utente l'immissione di più informazioni, perciò è opportuno visualizzare delle frasi esplicative; a tale scopo facciamo precedere le istruzioni `scanf` da appropriate `printf`.

```
printf("Valore base: ");

scanf("%d", &base);
```

L'argomento di `printf` è semplicemente una costante, quindi deve essere racchiuso tra doppi apici. Non è necessario neanche richiedere un salto a linea nuova mediante un **`\n`**, in quanto l'immissione può avvenire sulla stessa riga.

Quello che apparirà all'utente in fase di esecuzione del programma sarà

```
Valore base: _
```

In questo istante l'istruzione `scanf` attende l'immissione di un valore. Se l'utente digita `15` seguito da <Invio>.

```
Valore base: 15 <Invio>
```

questo dato verrà assegnato alla variabile `base`

Analogamente possiamo modificare il programma per l'immissione dell'altezza e magari aggiungere un'intestazione che spieghi all'utente cosa fa il programma, come nel listato seguente:

```
#include <stdio.h>

/* Calcolo area rettangolo */

main()

{

    /* dichiarazione delle variabili */

    int base,altezza,area;


    /* input dei dati */

    printf("Calcolo AREA RETTANGOLO \n \n");

    printf("Valore base: ");

    scanf("%d",&base);

    printf("\nValore altezza: ");

    scanf("%d",&altezza);


    /* parte elaborativa */

    area = base*altezza;


    /* output dei risultati */
```

```
    printf("\nBase: %d Altezza: %d\n",base,altezza);

    printf("Area: %d",area);


}/* fine programma*/
```

Vediamo l'esecuzione del programma nell'ipotesi che l'utente inserisca i valori 10 e 13.

```
Calcolo AREA RETTANGOLO

Valore base: 10 <Invio>

Valore altezza: 13 <Invio>

Base: 10 Altezza: 13

Area: 130
```

Notare l'uso del *newline* per controllare il modo in cui il programma visualizzerà i suoi risultati. I due **`\n`** della prima `printf`, per esempio, servono: il primo per passare ad una nuova linea, il secondo per lasciare una linea vuota.

Con una sola istruzione di input è possibile acquisire più di un valore, per cui i due input dell'esempio precedente, avrebbero potuto essere sostituiti da:

```
printf("Introdurre Base e Altezza separati da una virgola\n");

scanf("%d,%d",&base,&altezza);
```

In questo caso, quando il programma viene eseguito, alla richiesta di input si risponderà con due numeri (che saranno assegnati rispettivamente a `base` e ad `altezza`), separati da una virgola. Il motivo della presenza della virgola è dovuto alla specifica presente nella `scanf`: infatti ci sono due `%d` separati da una virgola.

inizio

Istruzione *if*

Quando si desidera eseguire un'istruzione al presentarsi di una certa condizione, si utilizza l'istruzione `if`.

Per esempio, se si vuole visualizzare il messaggio "`il valore attuale di i è minore di 100`" solamente nel caso in cui il valore della variabile intera `i` è minore di `100`, si scrive:

```
if( i<100 )
    printf("\n il valore attuale di i è minore di 100");
```

La sintassi dell'istruzione if è:

```
if( espressione )
   istruzione;
```

dove la valutazione di espressione controlla l'esecuzione di istruzione: se espressione è vera viene eseguita istruzione.

Nell'esempio seguente il programma richiede un numero all'utente e, se tale numero è minore di `100`, visualizza un messaggio.

```
#include <stdio.h>

/* Esempio utilizzo if */

main()

{

    /* dichiarazioni di variabili */

    int i;


    /* input dei dati */

    scanf("%d",&i);


    /* parte elaborativa */

    if (i<100)

        printf("\n minore di 100");


}/* finr programma */
```

L' espressione `i<100` è la *condizione logica* che controlla l'istruzione di stampa e pertanto la sua valutazione potrà restituire soltanto uno dei due valori booleani **vero** o **falso** che in C corrispondono rispettivamente ai valori interi **uno** e **zero**.È appunto per tale ragione che l'assegnamento `a=i<100`, è del tutto lecito. Viene infatti valutata l'espressione logica `i<100`, che restituisce `1` (vero) se `i` è minore di `100` e `0` (falso) se `i` è maggiore uguale a `100`: il risultato è dunque un numero intero che viene assegnato alla variabile `a`.

L'operatore `!=` corrisponde a *diverso da*, per cui l'espressione `a!=0` significa: il valore di `a` è diverso da zero.

Chiedersi se il valore di `a` è diverso da zero è lo stesso che chiedersi se il valore di `a` è vero, il che, in C, corrisponde al controllo eseguito per *default* (effettuato in mancanza di differenti indicazioni), per cui avremmo anche potuto scrivere

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

a = i<100;

if (a)

    printf("\n minore di 100");
```

La sintassi completa dell'istruzione i f è la seguente:

```
if(espressione)

   istruzione1;

[else

   istruzione2;]
```

dove la valutazione di espressione controlla l'esecuzione di istruzione1 e istruzione2: se espressione è vera viene eseguita istruzione1, se è falsa viene eseguita istruzione2.

Nell'esempio anteriore è stato omesso il ramo `else`: il fatto è del tutto legittimo in quanto esso è opzionale, come evidenziato dalle parentesi quadre presenti nella forma sintattica completa.

La tabella seguente mostra gli operatori utilizzabili nell'istruzione if :

Operatore	Esempio	Risultato
`!`	`!a`	(NOT logico) `1` se `a` è `0`, altrimenti `0`
`<`	`a<b`	`1` se `a<b`, altrimenti `0`
`<=`	`a<=b`	`1` se `a<=b`, altrimenti `0`
`>`	`a>b`	`1` se `a>b`, altrimenti `0`
`>=`	`a>=b`	`1` se `a>=b`, altrimenti `0`
`==`	`a==b`	`1` se `a` è uguale a `b`, altrimenti `0`
`!=`	`a!=b`	`1` se `a` non è uguale a `b`, altrimenti `0`
`&&`	`a&&b`	(AND logico) `1` se `a` e `b` sono veri, altrimenti `0`
`||`	`a||b`	(OR logico)`1` se `a` è vero, (`b` non è valutato), `1` se `b` è vero, altrimenti `0`

È opportuno notare che, nel linguaggio C, il confronto dell'eguaglianza fra i valori di due variabili viene effettuato utilizzando il doppio segno ==. Si confrontino i seguenti due frammenti:

Programma **A**

```
if( a==b )
   printf("Sono uguali\n");
```

Programma **B**

```
If( a=b )
   printf("Valore non zero\n");
```

Il programma **A confronta** il contenuto della variabile `a` ed il contenuto della variabile `b`: se sono uguali stampa la frase specificata.

Il programma **B assegna** ad `a` il valore attualmente contenuto in `b` e verifica se è diverso da zero: in tal caso stampa la frase specificata.

Una ulteriore osservazione va fatta a proposito degli operatori logici `!` (NOT logico), `&&` (AND logico) e `||` (OR logico) che vengono usati per mettere assieme più condizioni. Es.

```
if (a>5 && a<10)
Printf("a compreso fra 5 e 10");
```

```
if (a<2 || a>10)
printf("a può essere <2 oppure >10");
```

inizio

Istruzioni composte

L'istruzione composta, detta anche *blocco*, è costituita da un insieme di istruzioni inserite tra parentesi graffe che il compilatore tratta come se fosse un'istruzione unica.

Un'istruzione composta può essere scritta nel programma **dovunque** possa comparire un'istruzione semplice. Si noti la differenza di esecuzioni dei due frammenti di programma seguenti:

Programma **A**

```
if (a>100)
printf("Prima frase \n");
printf("Seconda frase \n");
```

Programma **B**

```
if (a>100) {
printf("Prima frase \n");
printf("Seconda frase \n");
};
```

Il programma **A** visualizzerà "`Prima frase`" solo se `a` è maggiore di 100, "`Seconda frase`" verrà visualizzato in ogni caso: *la sua visualizzazione prescinde infatti dalla condizione*.

Il programma **B**, qualora `a` non risulti maggiore di 100, non visualizzarà alcuna frase: le due `printf` infatti sono raggruppate in un *blocco la cui esecuzione è vincolata dal verificarsi della condizione*.

Un blocco può comprendere anche una sola istruzione. Ciò può essere utile per aumentare la chiarezza dei programmi: l'istruzione compresa in una `if` può essere opportuno racchiuderla in un blocco anche se è una sola. In tal modo risulterà più evidente la dipendenza dell'esecuzione della istruzione dalla condizione.

inizio

l'operatore *?*

L'operatore "`?`" ha la seguente sintassi:

```
espr1 ? espr2 : espr3;
```

Se espr1 è **vera** restituisce espr2 **altrimenti** restituisce espr3.

Si può utilizzare tale operatore per assegnare, condizionatamente, un valore ad una variabile. In questo modo può rendere un frammento di programma meno dispersivo:

Programma **A**

```
if (a>100)
sconto=10;
else
sconto=5;
```

Programma **B**

```
sconto=(a>100 ? 10 : 5);
```

inizio

Cicli e istruzione *while*

Le strutture cicliche assumono nella scrittura dei programmi un ruolo fondamentale, non fosse altro per il fatto che, utilizzando tali strutture, si può istruire l'eleboratore affinché esegua azioni ripetitive su insiemi di dati diversi: il che è, tutto sommato, il ruolo fondamentale dei sistemi di elaborazione.

È in ragione delle suddette considerazioni che i linguaggi di programmazione mettono a disposizione del programmatore vari tipi di cicli in modo da adattarsi più facilmente alle varie esigenze di scrittura dei programmi. La prima struttura che prendiamo in considerazione è il ciclo `while` (ciclo iterativo con *controllo in testa*):

```
while(esp)
   istruzione
```

Viene verificato che esp sia vera, nel qual caso viene eseguita istruzione. Il ciclo si ripete fintantoché esp risulta essere vera.

Naturalmente, per quanto osservato prima, istruzione può essere un blocco e, anche in questo caso, può essere utile racchiudere l'istruzione in un blocco anche se è una sola.

Scriviamo, a titolo di esempio di uso del ciclo `while`, un frammento di programma che calcola la somma di una serie di valori positivi immessi dall'utente:

```
...
somma = 0;
printf("\n Inserisci un intero positivo:");
scanf("%d", &numero);
while(numero)
{
    somma += numero;
    printf("\n Inserisci un intero positivo:");
    scanf("%d", &numero);
}
```

In questo caso il controllo all'inizio del ciclo garantisce la fine della elaborazione non appena la variabile `numero` assume valore zero: il ciclo viene ripetuto mentre `numero` è diverso da zero. L'input, fuori ciclo, del primo numero da elaborare permette di impostare la condizione di controllo sul ciclo stesso. Il totalizzatore `somma` *cumula* tutti i valori provenienti da input.

inizio

Cicli e istruzione *for*

L'istruzione `for` viene utilizzata tradizionalmente per codificare cicli a contatore: istruzioni cicliche cioè che devono essere ripetute un numero definito di volte. Il formato del costrutto `for` è il seguente:

```
for(esp1; esp2; esp3)
   istruzione;
```

Si faccia attenzione ai punti e virgola posti tra parentesi. Il ciclo inizia con l'esecuzione di esp1 (*inizializzazione del ciclo*) la quale non verrà più eseguita. Quindi viene esaminata esp2 (*condizione di controllo del ciclo*). Se esp2 risulta vera, viene eseguita istruzione, altrimenti il ciclo non viene percorso neppure una volta. Successivamente viene eseguita esp3 (*aggiornamento*) e di nuovo valutata esp2 che se risulta essere vera dà luogo ad una nuova esecuzione di istruzione. Il processo si ripete finché esp2 risulta essere vera. Quando esp2 risulterà falsa il ciclo avrà termine.

Se supponiamo di voler ottenere la somma di tre numeri interi immessi dall'utente, si può scrivere:

```
...
somma = 0;
for(i = 1; i <= 3; i++)
{
    scanf("%d", &numero);
    somma += numero;
}
```

Il programma per prima cosa assegna il valore `1` alla variabile `i` (la prima espressione del `for`), si controlla se il valore di `i` è non superiore a `3` (la seconda espressione) e poiché l'espressione risulta vera verranno eseguite le istruzioni inserite nel ciclo (l'input di `numero` e l'aggiornamento di `somma`). Terminate le istruzioni che compongono il ciclo si esegue l'aggiornamento di `i` così come risulta dalla terza espressione contenuta nel `for`, si ripete il controllo contenuto nella seconda espressione e si continua come prima finché il valore di `i` non rende falsa la condizione.

Questo modo di agire del ciclo `for` è quello comune a tutti i cicli di questo tipo messi a disposizione dai compilatori di diversi linguaggi di programmazione. Il linguaggio C mette a disposizione delle opzioni che espandono abbondantemente le potenzialità del `for` generalizzandolo in maniera tale da comprendere, per esempio, come caso particolare il ciclo `while`. Il frammento di programma per la somma di una serie di numeri positivi, scritto in precedenza, potrebbe, per esempio, essere riscritto:

```
...
printf("\n Inserisci un intero positivo:");
scanf("%d", &numero);
for(somma=0;numero;)
{
    somma += numero;
    printf("\n Inserisci un intero positivo:");
    scanf("%d", &numero);
}
```

Il ciclo esegue l'azzeramento di `somma` (che verrà eseguito una sola volta) e subito dopo il controllo se il valore di `numero` è diverso da zero e, in questo caso, verranno eseguite le istruzioni del ciclo. Terminate le istruzioni, poiché manca la terza espressione del `for`, viene ripetuto il controllo su `numero`.

L'inizializzazione di `somma` avrebbe potuto essere svolta fuori dalla `for`: in tal caso sarebbe mancata anche la prima espressione.

Poiché, nel linguaggio C, ogni ciclo `while` può essere codificato utilizzando un ciclo `for` e viceversa, è bene tenere presente che la scelta del tipo di codifica da effettuare va sempre fatta in modo da ottenere la massima **chiarezza** e **leggibilità** del programma.

inizio

Cicli e istruzione *do-while*

L'uso della istruzione `while` prevede il test sulla condizione all'inizio del ciclo stesso. Ciò vuol dire che se, per esempio, la condizione dovesse risultare falsa, le istruzioni facenti parte del ciclo verrebbero saltate e non verrebbero eseguite nemmeno una volta.

Quando l'istruzione compresa nel ciclo deve essere comunque eseguita almeno una volta, è più comodo utilizzare il costrutto:

```
do
   istruzione;
while(espr);
```

In questo caso viene eseguita istruzione e **successivamente** controllato se espr risulta vera, nel qual caso il ciclo viene ripetuto.

Come sempre l'iterazione può comprendere una istruzione composta.

È bene precisare che in un blocco `for`, `while` o `do...while`, così come nel blocco `if`, può essere presente un numero qualsiasi di istruzioni di ogni tipo ivi compresi altri blocchi `for`, `while` o `do...while`. I cicli possono cioè essere *annidati*.

Tipi di dati e modificatori di tipo

Negli esempi trattati fino a questo punto tutte le variabili sono state dichiarate di tipo `int`. È bene però chiarire che esistono anche altri tipi di dichiarazione delle variabili in modo che, le stesse, possano contenere valori di natura diversa. D'altra parte ciò è facilmente intuibile: basta pensare alle limitazioni dettate appunto dal tipo `int`. In realtà i vari linguaggi di programmazione permettono di definire le variabili in diversi modi; ciò in relazione al tipo di dato da contenere e alle esigenze specifiche del tipo di problemi cui si rivolge il linguaggio stesso.

Nel linguaggio C esistono cinque tipi di dati primari: **carattere**, **intero**, **reale**, **doppia precisione** e **indefinito**. Le parole chiavi utilizzate per dichiarare variabili di questi tipi sono: `char`, `int`, `float`, `double` e `void`. La dimensione del dato contenibile nella variabile dipende dallo spazio riservato in memoria per ogni tipo di dichiarazione e dalla modalità di conservazione.

Per quanto riguarda il tipo `void` (tipo indefinito), si tratterà di ciò in un secondo tempo. Per il momento l'attenzione sarà rivolta agli altri tipi.

I tipi di dati primari, eccetto il tipo `void`, ammettono dei modificatori di tipo. Tali modificatori (`signed`, `unsigned`, `long`, `short`) sono impiegati per adattare con maggiore precisione i tipi di dati fondamentali alle esigenze del programmatore.

Nella tabella successiva sono elencati le varie combinazioni più comunemente usate con un esempio di utilizzo (per i valori ammessi si fa quì riferimento alla implementazione del C nei PC con processori a 32 bit):

Tipo	Dim. in bit	Valori ammessi	Utilizzo
`char`	8	da -128 a 127	Numeri piccoli e caratteri ASCII
`unsigned char`	8	da 0 a 255	Piccoli numeri e il set di caratteri del PC
`short int`	16	da -32.768 a +32.767	valori booleani (0/1)
`int`	32	da -2.147.483.648 a 2.147.483.647	contatori, accumulatori, etc...
`unsigned int`	32	da 0 a 4.294.967.395	Numeri e cicli
`long int`	32	da -2.147.483.648 a 2.147.483.647	come gli int
`float`	32	da $3,4x10^{-38}$ a $3,4x10^{38}$	Notazione Scientifica (7 cifre di precisione)
`double`	64	da $1,7x10^{-308}$ a $1,7x10^{308}$	Notazione Scientifica (15 cifre di precisione)

È bene fare qualche osservazione sulla tabella:

- I tipi `signed` (`char`, `int`, `long`) conservano in memoria dati utilizzando il metodo del **complemento a 2**
- Il tipo `float` e il tipo `double` sono rappresentazioni **floating-point**: vengono comunemente indicati, rispettivamente, come numeri a *singola precisione* e a *doppia precisione*
- Si noti che il tipo `char` viene utilizzato sia per conservare piccoli numeri che per conservare il set di caratteri, cioè dati apparentemente completamente diversi fra di loro dal punto di vista della

elaborazione. La cosa è meno strana di quello che può sembrare a prima vista, basta riflettere sulla circostanza che, anche se si tratta di caratteri, la conservazione in memoria avviene associando ad ogni carattere una configurazione binaria e, quindi, un numero. Si può allora considerare la stessa configurazione come numero o come carattere corrispondente a quella configurazione binaria: la configurazione `0100.0001` (secondo per esempio la codifica ASCII) può essere il numero `65` come anche il carattere '`A`'. Tutto ciò può tornare utile per certi tipi di elaborazione sui caratteri.

- La specificazione `int`, se accompagnata da un modificatore, può essere omessa. Si può quindi dichiarare una variabile indifferentemente `long int` o solo `long`

Per ogni tipo è prevista dal C una stringa di controllo per l'output formattato della `printf` (per il tipo `int`, come si ricorderà, tale stringa è `%d`): per la conversione del tipo `char` la stringa `%c`, per quella di tipo `float` o `double` la stringa `%f`, `%l` sta per `long` e `%u` per `unsigned`, la stringa `%e` fa in modo che il valore della variabile mandata in output sia visualizzato con il formato esponenziale.

Nelle assegnazioni è spesso necessario specificare meglio il valore da assegnare:

```
..

long a;

double b;

char c;

..

a = 12L;

b = 15.0;

c = 'A';
```

Il valore `12` andrebbe pure in una variabile di tipo `int`, il suffisso `L` fa in modo che venga espanso in un `long`.

Nelle assegnazioni ai tipi in virgola mobile è necessario specificare sempre la parte decimale anche quando, come nel caso precedente, non sarabbe indispensabile.

Nelle assegnazioni alle variabili di tipo `char` il carattere da assegnare deve essere racchiuso fra apici.

Esaminiamo adesso un programma che trasforma un carattere minuscolo nella sua rappresentazione maiuscola. Le righe più significative sono commentate da etichette numeriche cui si farà riferimento poi per alcune osservazioni:

```
/*

Conversione di un carattere da minuscolo a maiuscolo

Vale per la codifica ASCII

*/

#include <stdio.h>

#define SCARTO 32 /*1*/

main()
```

```
{
    /* dichiarazioni variabili */
    char min,mai;

    /* input dati */
    printf("\n Conversione minuscolo-maiuscolo");
    printf("\n Introduci un carattere minuscolo");
    scanf("%c",&min); /*2*/

    /* parte elaborativa e output risultati */
    if (min>=97 && min<=122) /*3*/
    {
        mai = min-SCARTO; /*4*/
        printf("\n Rappresentazione maiuscola %c",mai); /*5*/
        printf("\n Codice ASCII %d",mai); /*6*/
    }
    else
        printf("\n Carattere non convertibile");

}/* fine programma */
```

Nella riga con etichetta `1` è definita la costante `SCARTO`. Il valore `32` dipende dal fatto che, nel codice ASCII, tale è la distanza fra le maiuscole e le minuscole (es. '`A`' ha codice `65`, '`a`' ha codice `97`).

Nella riga con etichetta `2` si effettua l'input del carattere da elaborare. Si noti l'uso di `%c` per indicare che si tratta appunto di un carattere.

Nella riga con etichetta 3 si controlla, utilizzando la sua rappresentazione numerica, se il carattere immesso rientra nei limiti della codifica ASCII delle lettere minuscole. Le lettere minuscole, in ASCII, hanno codice compreso fra `97` (la lettera minuscola `a`) e `122` (la lettera minuscola `z`). Il confronto poteva anche essere fatto sulla rappresentazione alfanumerica:

```
if (min>='a' && min<='z')
```

Nella riga con etichetta `4` si effettua in pratica la trasformazione in maiuscolo. Al codice numerico associato al carattere viene sottratto il valore `32`. In questo caso è utilizzato il codice numerico del carattere. Si noti che in questo contesto ha senso assegnare ad un `char` il risultato di una sottrazione (operazione numerica).

Nella riga con etichetta `5` si effettua l'output di `mai` inteso come carattere (è usato `%c`), laddove nella riga `6` si effettua l'output del suo codice numerico (è usato `%d`).

Le direttive di conversione possono avere varianti che specificano l'ampiezza dei campi, il numero di cifre decimali e un indicatore per l'allineamento del margine sinistro.

Un intero posto fra il segno di percentuale e il carattere di conversione indica l'ampiezza minima del campo (vengono aggiunti spazi per assicurare che questa ampiezza sia garantita, se, invece degli spazi si voglio tanti zeri, basta aggiungere uno zero prima dello specificatore di ampiezza). La direttiva `%05d`, per esempio, usa zeri come caratteri riempitivi in modo da portare la lunghezza totale a cinque caratteri. Per specificare il numero di cifre decimali di un numero in virgola mobile, si scrive, dopo lo specificatore di ampiezza, un punto seguito dal numero di cifre che si vuole siano stampate. La direttiva `%10.4f`, per esempio, stampa un numero di non meno di 10 cifre con 4 cifre dopo la virgola. Si può inoltre fare in modo che la stampa sia allineata a sinistra (per default lo è a destra) ponendo un segno meno: `%-10.2f`.

inizio

Il costrutto cast

A volte può interessare effettuare dei cambiamenti al volo per conservare i risultati di determinate operazioni in variabili di tipo diverso principalmente quando si va da un tipo *meno capiente* ad un tipo *più capiente*. In tali casi il linguaggio C mette a disposizione del programmatore un costrutto chiamato ***cast***.

Es:

```
main()

{

    int uno, due;

    float tre;


    uno = 1;

    due = 2;

    tre = uno/due;

    printf("%f",tre);

}
```

Tale programma nonostante le aspettative (dettate dal fatto che la variabile `tre` è dichiarata `float`) produrrà un risultato nullo. Infatti la divisione viene effettuata su due valori di tipo `int`, il risultato viene conservato temporaneamente in una variabile di tipo `int` e, solo alla fine, conservato in una variabile `float`. È evidente, a questo punto, che la variabile `tre` conterrà solo la parte intera (cioè `0`).

Affinchè la divisione produca il risultato atteso, è necessario avvisare il C di convertire il risultato intermedio prima della conservazione definitiva nella variabile di destinazione.

Tutto ciò è possibile utilizzando il costrutto in questione che ha la seguente sintassi:

```
(nome-di-tipo) espressione
```

Utilizzando questo costrutto, il nostro programma, si scriverebbe così:

```
main()
```

```
{

    int uno, due;

    float tre;


    uno = 1;

    due = 2;

    tre = ( (float)uno )/due;

    printf("%f",tre);

}
```

In questo caso il programma fornisce il risultato atteso. Infatti il quoziente viene calcolato come `float` e quindi, dopo, assegnato alla variabile `tre`.

In definitiva il costrutto *cast* forza una espressione a essere di un tipo specifico (nel nostro caso una divisione intera viene forzata a fornire un risultato di tipo virgola mobile).

inizio

Array a una dimensione

L' **array** (in italiano conosciuto anche come **vettore**), è la prima struttura di dati trattata in Informatica. È quella più comunemente usata ed è quella che sta a fondamento di quasi tutte le altre.

Un array è un insieme di variabili dello stesso tipo cui è possibile accedere tramite un nome comune e referenziare uno specifico elemento tramite un indice. Possiamo pensare all'array come ad un insieme di cassetti numerati: per poter accedere al contenuto di un cassetto deve essere specificato il numero ad esso associato (l'**indice**).

Nelle variabili semplici per accedere al valore contenuto in esse è necessario specificare il nome ed, inoltre, una variabile con un valore diverso avrà un nome diverso. Nell'array esiste un nome che però stavolta identifica l'array come struttura (il nome farà riferimento all'insieme dei cassetti dell'esempio precedente: questo concetto sarà chiarito meglio più avanti). I singoli elementi dell'array verranno referenziati specificando la loro posizione relativa all'interno della struttura (l'indice): l'elemento a cui ci si riferisce dipende dal valore assunto dall'indice.

Gli elementi dell'array vengono allocati in posizioni di memoria adiacenti e, per tale motivo, è necessario dichiarare un array prima del suo uso in modo che il compilatore possa riservare in memoria lo spazio necessario per conservarlo.

Il formato generale della dichiarazione di un array a una dimensione è:

```
tipo nome_variabile[dimensione];
```

dove tipo è il tipo base dell'array, ovvero il tipo di ogni elemento dell'array; e il numero di elementi dell'array è definito da dimensione.

Il seguente esempio dichiara un array chiamato `numero` contenente un massimo di 10 elementi interi:

```
int numero[10];
```

Poiché il C assegna gli indici agli elementi di un array a partire da 0, gli elementi dell'array di interi appena dichiarato vanno da numero[0] a numero[9]. Quindi, per quanto osservato precedentemente: numero è il nome della struttura, identifica l'insieme degli elementi cioè 10 numeri interi, numero[0], numero[5] individuano il primo ed il sesto numero della struttura. Se si utilizza numero[i], l'elemento a cui ci si riferisce dipende dal valore assunto dalla variabile i che deve essere di tipo int.

Essendo poco abituati a concepire un elemento in posizione zero potrebbe essere opportuno, almeno inizialmente, sistemare gli elementi nell'array a cominciare dalla posizione 1: si sprecherà memoria riservata ad una posizione non utilizzata, ma i programmi potrebbero risultare più leggibili.

È bene ricordare che, seguendo tale convenzione, un array dichiarato contenente un massimo teorico di 10 elementi, può conservare gli stessi dalla posizione 1 alla posizione 9 (di fatto non potrà contenere più di nove elementi).

Il programma di esempio carica il nostro array con numeri provenienti dall'input:

```
main()

{

    int numero[10], conta;

    for (conta=1;conta<=9;conta++)

        scanf("%d",&numero[conta]);

}
```

Si noti nel programma di esempio l'uso della variabile conta: con una sola istruzione di input ripetuta il programma acquisisce valori da assegnare ai vari elementi dell'array. Il primo input viene effettuato su numero[1] (conta la prima volta vale 1), il secondo viene effettuato su numero[2] (conta vale 2) e così via fino al nono.

Questo tipo di array vengono detti ad una dimensione perchè per individuare un elemento all'interno della struttura, basta specificare un solo indice.

Il linguaggio C non esegue una verifica sui limiti dell'array, pertanto esiste la possibilità di superarli. Se ciò si verifica, per esempio in un assegnamento, si corre il rischio di alterare un'altra variabile o un pezzo di codice del programma. In altri termini si può utilizzare un array di dimensione n oltre i suoi limiti senza che vengano segnalati errori di compilazione o di esecuzione, ma con il risultato che probabilmente sarà il programma a mostrare dei malfunzionamenti.

Come caso di applicazione di un array numerico consideriamo il seguente esempio: le temperature rilevate nel corso della giornata in una determinata località sono conservate in un vettore, si richiede di scrivere un programma che, acquisita una determinata temperatura, determini se nella località considerata si è avuta nel corso della giornata tale temperatura e, in caso positivo, in quale rilevazione è registrata tale temperatura.

In termini informatici si tratta di effettuare una *scansione sequenziale* di un vettore per verificare se tale vettore contiene un determinato valore e, in caso positivo, se ne vuole conoscere la posizione.

Nelle righe più interessanti è presente un commento numerico per le osservazioni successive.

```
/* Ricerca un elemento in vettore e restituisce la posizione */

#include <stdio.h>

main()
```

```
{
    int temp[8],temprif; /*1*/
    int i,posiz;

    /* Acquisizione delle temperature rilevate */
    for (i=0;i<=7;i++)
    {
        printf("\nRilevazione temperatura n° %d ",i);
        scanf("%d",&temp[i]);
    }

    /* input chiave di ricerca */
    printf("\nTemperatura da ricercare ");
    scanf("%d",&temprif);

    /* Scansione del vettore delle temperature */
    posiz = -1; /*2*/
    for (i=0;i<=7 && posiz==-1;i++) /*3*/
    {
        if (temprif==temp[i])
            posiz = i; /*4*/
    }

    /* Risultati della ricerca */
    if (posiz == -1) /*5*/
        printf("\nTemperatura non rilevata");
    else
        printf("\nTemperatura acquisita nella rilevazione %d",posiz);

}/* fine programma */
```

Nella riga 1 si dichiara un array di 8 interi per conservare le rilevazioni delle temperature e una variabile per conservare la temperatura da cercare.

Nella riga 2 si inizializza una variabile che conterrà la posizione nella quale si troverà la temperatura cercata. Il valore −1 dipende dal fatto che le posizioni ammesse vanno dal valore 0 in su e, quindi, per indicare un valore che non indichi una posizione valida occorrerà utilizzare per esempio un numero negativo.

Nella riga 3 inizia il ciclo di scansione dell'array alla ricerca della temperatura di riferimento. Il controllo di fine ciclo si basa su due eventi che devono verificarsi contemporaneamente: elementi non finiti (`i<=7`) e temperatura non trovata (`posiz==-1`). Se infatti la temperatura cercata viene ritrovata è inutile continuare ancora ad esaminare gli altri elementi dell'array. Più avanti si esaminerà un modo diverso di uscita anticipata da un ciclo a contatore.

Se la temperatura ricercata è presente nell'array allora, nella riga 4, se ne conserva la posizione nella varabile `posiz`.

La riga 5 seleziona il caso temperatura non trovata (volere di `posiz` non modificato) dalla rilevazione ritrovata.

L'algoritmo sviluppato si basa sull'ipotesi che, se c'è, la temperatura ricercata si presenta una sola volta. Per togliere tale limitazione è necessario predisporre un vettore per contenere le posizioni delle varie ricorrenze. Le modifiche da apportare al programma saranno:

- Nella 1 dichiarare un vettore per conservare le posizioni. Per esempio `postemp[8]`
- Nella condizione di uscita del ciclo presente nella 3 occorre togliere il test su `posiz`. Qui infatti se si trova la temperatura cercata occorre esaminare tutto l'array alla ricerca delle eventuali altre ricorrenze.
- La riga 4 verrebbe modificata in: `postemp[++posiz]=i;` bisogna cioè conservare la posizione `i` nell'array `postemp` nella posizione successiva a quella indicata da `posiz` (si tenga presente che il valore iniziale è −1). Il contatore `posiz` va quindi *prima* incrementato e *poi* utilizzato come indice dell'array.
- Naturalmente la struttura condizionale 5 viene modificata per permettere di mandare in output l'array `postemp`. Si può utilizzare un ciclo a contatore simile a quello utilizzato per l'input delle temperature: l'indice `i` varierà fra 0 e `posiz` e il ciclo conterrà una `printf` per la stampa dell'elemento dell'array `postemp`.

inizio

Stringhe

In C uno degli utilizzi degli array ad una dimensione riguarda la conservazione in memoria di stringhe di caratteri: queste in C sono array di tipo carattere la cui fine è segnalata da un carattere ***null*** (carattere terminatore), indicato come `'\0'`.

Il carattere *null* è il primo codice ASCII corrispondente al valore binario `00000000` e non ha niente a che vedere con il carattere `0` che ha, in ASCII, codice binario `00110000`.

```
char a[10];
```

dichiara un vettore costituito da un massimo di dieci caratteri e:

```
char frase[] = "Oggi c'è il sole";
```

dichiara l'array monodimensionale `frase` il cui numero di caratteri è determinato dalla quantità di caratteri presenti fra doppi apici più uno (il carattere *null* che chiude la stringa).

È importante notare la differenza tra le due assegnazioni:

```
char a = 'r';

char b[] = "r";
```

Nel primo caso viene assegnato il solo carattere `r`, nel secondo la stringa `r\0`. In definitiva: se si vuole fare riferimento ad un solo carattere, questo deve essere racchiuso fra apici singoli; se, invece, si vuole fare riferimento ad una stringa, occorre racchiuderla fra doppi apici.

Sono disponibili parecchie funzioni specifiche per il trattamento delle stringhe nonostante il linguaggio C non possieda un dato di tipo stringa:

gets e puts

Le funzioni `gets` e `puts` sono funzioni specializzate nell'input e output di una stringa. Il programma seguente riceve in input una stringa e, subito dopo, la rimanda in output. Notare che in queste istruzioni si fa riferimento alla stringa nel suo complesso e, quindi, viene utilizzato il nome:

```
#include <stdio.h>

main()

{

    char s[80]; /* dichiara la stringa */

    gets(s); /* riceve l'input */

    puts(s); /* fornisce l'output */

}
```

Altre funzioni per il trattamento delle stringhe sono disponibili, per usi particolari, a patto che si includa nel programma la linea: `#include <string.h>`.

strcpy

Questa funzione copia una stringa in un'altra. Può essere utilizzata, per esempio, per assegnare un valore a una stringa. La sua sintassi è:

```
strcpy(s1,s2);
```

Si può immaginare che tale funzione equivale, come effetto, all'assegnamento nelle variabili di altro tipo (in altri termini è come se si scrivesse: `s1=s2` assegna, cioè, alla stringa `s1` il valore contenuto in `s2`. In realtà, come si vedrà più avanti, tale istruzione è valida ma assume altro significato).

strcat, strlen e strcmp

La funzione:

```
strcat(s1,s2);
```

concatena la stringa `s2` alla fine della stringa `s1`:

```
strcpy(s1,"buon"); /* stringa specificata in s1 */

strcpy(s2,"giorno"); /* lo stesso per s2 */
```

```
strcat(s1,s2); /* s1 conterrà "buongiorno" */
```

La funzione:

```
strlen(s1);
```

restituisce un valore numerico che rappresenta la lunghezza della stringa `s1`.

La funzione:

```
strcmp(s1,s2);
```

può essere utilizzata per effettuare comparazioni sul contenuto di due stringhe. In particolare tale funzione:

- Restituisce un valore positivo se vale `s1>s2`
- Restituisce un valore negativo se vale `s1<s2`
- Restituisce `0` se `s1` ed `s2` sono uguali

Per tenere conto mnemonicamente di tali valori, basta pensare al confronto come ad una sottrazione: fra l'altro ciò è non è molto distante da quello che avviene in effetti. Se da `s1` si sottrae `s2` si avrà un valore positivo nel caso `s1>s2`, negativo se `s1<s2` e nullo se sono uguali

inizio

La scelta multipla: istruzione *else-if*

Può essere necessario, nel corso di un programma, variare l'elaborazione in seguito a più condizioni. In questi casi il linguaggio C fornisce la struttura:

```
if(espressione)
   istruzione;
else if(espressione)
   istruzione;
..
else
   istruzione;
```

In questo caso le espressioni vengono valutate nell'ordine in cui sono scritte: se è vera la prima espressione si esegue l'istruzione specificata e poi si passa all'istruzione successiva alla chiusura della struttura. Se la prima condizione non è vera si passa ad esaminare la seconda e così via. Si tenga presente che:

- Ci possono essere tante `else if` quante ne occorrono per l'elaborazione richiesta. Per ogni elaborazione l'istruzione eseguita è solo quella soggetta alla condizione verificata
- La clausola `else` finale, se presente (in quanto, come l'equivalente nella `if` già vista in precedenza, può mancare) gestisce l'istruzione eseguita quando nessuna delle espressioni specificate è verificata.

Al fine di mostrare un esempio di alcune delle istruzioni esposte ultimamente scriviamo un programma che, data una espressione algebrica, calcoli quante parentesi graffe, quadre e tonde sono contenute nella espressione stessa. Le righe più significative sono etichettate per permettere un commento successivo:

```
/*
Conta i vari tipi di parentesi contenute in una espressione algebrica
(non fa distinzione fra parentesi aperte e chiuse)
*/
#include <stdio.h>
main()
{
    /* dichiarazione variabili */
    char espress[30]; /*1*/
    int pargraf,parquad,partond;
    int c;

    /* input dati */
    printf("\nEspressione algebrica ");
    gets(espress); /*2*/

    /* parte elaborativa */
    pargraf=parquad=partond=0;
    for(c=0;c<=29 && espress[c]!='\0';c++) /*3*/
    {
        if(espress[c]=='{' || espress[c]=='}') /*4*/
            pargraf++;
        else if(espress[c]=='[' || espress[c]==']') /*4*/
            parquad++;
        else if(espress[c]=='(' || espress[c]==')') /*4*/
            partond++;
    }/* fine ciclo for*/

    /* output risultati */
```

```
        printf("\nGraffe = %d",pargraf);

        printf("\nQuadre = %d",parquad);

        printf("\nTonde = %d",partond);

    }/* fine programma */
```

Nella riga con etichetta `1` si dichiara un vettore di tipo `char` e di lunghezza massima di `30` caratteri, cioè una stringa di `30` caratteri.

Nella riga con etichetta `2` si acquisisce dall'input la stringa (cioè l'espressione algebrica da elaborare).

Nella riga con etichetta `3` c'è l'inizio di un ciclo a contatore che servirà per la scansione della stringa: i caratteri verranno esaminati uno per volta per verificare se sono parentesi. Nella `gets` la stringa è vista nel suo complesso, qui si esaminano i singoli elementi dell'array cioè i caratteri che compongono la stringa. Si noti il controllo di fine ciclo: il contatore del ciclo, che qui assume il compito di indice del vettore, non può superare `29` (l'array ha `30` elementi al massimo) ed inoltre il carattere esaminato non può essere *null* (qualora lo fosse, la stringa sarebbe terminata). Le due condizioni devono essere verificate contemporaneamente: se una delle due non è verificata (superamento dei limiti fissati o fine stringa) l'elaborazione passa all'istruzione immediatamente dopo la chiusura del ciclo: la `printf` dopo la chiusura della parentesi (qui le parentesi sono state aggiunte per aumentare la leggibilità infatti, contenendo il ciclo una sola istruzione, non sarebbero indispensabili).

Nelle righe etichettate con `4` si esaminano i vari casi possibili (i tre tipi di parentesi). Il carattere esaminato può essere indifferentemente la parentesi aperta o quella chiusa e, a seconda del tipo, viene conteggiato nell'apposito contatore. Se il carattere esaminato non è una parentesi, mancando la clausola `else`, non viene preso in considerazione.

inizio

La scelta multipla: istruzioni *switch-case* e *break*

Esiste nel linguaggio C un'altra istruzione per codificare la scelta multipla. È necessario fare attenzione però alle diversità di comportamento. Sintatticamente la struttura si presenta in questo modo:

```
switch(espressione)

{

case valore: istruzione;

case valore: istruzione;

..

[default: istruzione;]

}
```

In pratica in questo caso vengono valutati i diversi valori che può assumere una espressione. Nelle varie istruzioni `case` si elencano i valori che può assumere espressione e che interessano per l'elaborazione. Valutata l'espressione specificata, se il valore non coincide con alcuno di quelli specificati, viene eseguita l'istruzione compresa nella clausola `default`. Occorre tenere presente che, sostanzialmente, i vari `case` esistenti agiscono da etichette: se espressione assume un valore specificato in un `case`, vengono eseguite le

istruzioni **a partire da quel punto in poi**. Il valore specificato in `case`, in definitiva, assume funzione di *punto di ingresso* nella struttura.

Se si vuole delimitare l'istruzione da eseguire a quella specificata nella `case` che verifica il valore cercato, occorre inserire l'istruzione `break` che fa proseguire l'elaborazione dall'istruzione successiva alla chiusura della struttura.

Per chiarire meglio il funzionamento della struttura riscriviamo il programma del conteggio delle parentesi di una espressione algebrica facendo uso della nuova struttura.

```
/*
Conta i vari tipi di parentesi contenute in una espressione algebrica
(non fa distinzione fra parentesi aperte e chiuse)
*/
#include <stdio.h>
main()
{
    /* dichiarazioni variabili */
    char espress[30];
    int pargraf,parquad,partond;
    int c;

    /* input dati */
    printf("\nEspressione algebrica ");
    gets(espress);

    /* parte elaborativa */
    pargraf=parquad=partond=0;
    for(c=0;c<=29 && espress[c]!='\0';c++)
    {
        switch(espress[c])  /*1*/
        {
        case '{':
        case '}': pargraf++  /* 2 */;
        break; /*3*/
```

```
            case '[':
            case ']': parquad++;/*2*/
            break; /*3*/

            case '(':
            case ')': partond++; /*2*/
            break;
            }/* fine switch */
        }/* fine ciclo for */

        /* output risultati */
        printf("\nGraffe = %d",pargraf);
        printf("\nQuadre = %d",parquad);
        printf("\nTonde = %d",partond);
    }/* fine programma */
```

Nella riga con etichetta `1` viene specificata l'espressione da valutare: `espress[c]` cioè il carattere estratto dall'espressione algebrica.

Nelle righe con etichetta `2` si esaminano i casi parentesi aperta o parentesi chiusa. I singoli valori sono seguiti dalla istruzione nulla (il solo carattere `;`) e, poiché l'elaborazione continua da quel punto in poi, sia che si tratti di parentesi aperta che di parentesi chiusa si arriva all'aggiornamento del rispettivo contatore.

Nelle righe con etichetta `3` si blocca l'esecuzione del programma altrimenti, per esempio, una parentesi graffa oltre che come graffa verrebbe conteggiata anche come quadra e tonda, una parentesi quadra verrebbe conteggiata anche come tonda. Si noti che, anche se sono presenti due istruzioni, non vengono utilizzate parentesi per delimitare il blocco: il funzionamento della `switch-case` prevede infatti la continuazione dell'elaborazione con l'istruzione successiva. L'ultima istruzione `break` è inserita solo per coerenza con gli altri casi. Inoltre se in seguito si dovesse aggiungere una istruzione `default`, il programma continuerebbe a dare risultati coerenti senza necessità di interventi se non nella parte da inserire.

inizio

Array a più dimensioni

Un array a più dimensioni è un insieme di elementi dello stesso tipo, ognuno referenziabile da un insieme di indici: tanti quante sono le dimensioni dell'array stesso. Se si ha un **array bidimensionale**, per esempio, occorrono **due indici** per identificare un elemento della struttura.

Il formato generale della dichiarazione di un array a più dimensioni è:

```
tipo nome_variabile[dimensione] [dimensione] [dimensione]..
```

Gli array bidimensionali (***matrici***), che sono quelli più comunemente utilizzati, possono essere pensati come un insieme di elementi posizionati in una tabella: il primo indice identifica la riga, il secondo la colonna. Un elemento viene identificato specificando la riga e la colonna in cui si trova.

Come esempio di uso di un array a due dimensioni scriviamo un programma che costruisce e poi visualizza la tavola pitagorica. Le righe più significative sono, al solito, corredate da etichette numeriche che serviranno per i commenti successivi:

```
/* Stampa su video la tavola pitagorica */

#include <stdio.h>

main()

{

    /* dichiarazione della tavola e degli indici per scorrerla */

    int tavpit[11][11]; /*1*/

    int i,j;



    /* Costruisce la tavola */

    for (i=1;i<=10;i++) /*2*/

    {

        for (j=1;j<=10;j++) /*3*/

        {

            tavpit[i][j]=i*j; /*4*/

        }/* fine ciclo for interno */

    }/* fine ciclo for esterno */



    /* Manda in output la tavola */

    for (i=1;i<=10;i++)

    {

        for (j=1;j<=10;j++)

            printf("%3d ",tavpit[i][j]); /*5*/

        printf("\n"); /*6*/

    }/* fine ciclo for esterno */

}/* fine programma */
```

Nella riga con etichetta `1` si dichiara una matrice di `11` righe ed `11` colonne.

Nella riga con etichetta 2 si inizia il ciclo per la scansione delle righe della tabella (l'indice i viene utilizzato per identificare la riga esaminata della tabella). L'indice viene inizializzato ad 1 non avendo interesse a visualizzare la riga formata da tutti valori nulli.

Nella riga con etichetta 3 si inizia un nuovo ciclo, interno al primo, per la scansione delle colonne. L'uso delle parentesi mette in evidenza il funzionamento di questo doppio ciclo: si esamina la prima riga (i=1) e quindi le varie colonne di quella riga (per i=1 l'indice J varia da 1 a 10) . Si passa poi alla riga successiva e così via.

Nella riga con etichetta 4 si assegna il valore all'elemento della tabella che sta nella riga i e nella colonna j: nel nostro caso sarà equivalente al prodotto fra i due indici.

Nella riga con etichetta 5 viene mandato in output l'elemento generico. La stringa di formattazione dell'output consente l'allineamento e l'incolonnamento dei numeri della tabella.

La printf della riga 6 manda in output semplicemente un *newline* che consente di passare a capo quando si tratta di stampare su video una nuova riga della tabella. La printf è infatti posizionata alla fine del ciclo controllato da j (quello che stampa le colonne di una riga), quindi dopo la stampa di tutte le colonne di una riga, e prima della chiusura del ciclo controllato da i (quello che stampa le righe) quindi prima di passare alla prossima riga.

inizio

Array di stringhe

Una applicazione interessante delle matrici è l'array di stringhe. La stringa è un array di caratteri e quindi potremmo dire che un array di stringhe è un array di array di caratteri. Per chiarire la gestione di questo tipo di array esaminiamo un programma che si occupa di cercare delle parole in un vocabolario e ci dica, per ognuna, se è contenuta nel vocabolario stesso

```
/* Cerca parole in un vocabolario */#include <stdio.h>

#include <string.h> /*1*/

main()

{

    /* dichiarazione array e indici */

    char vocab[10][20], parcerc[20]; /*2*/

    int i,trovata;


    /* Acquisisce parole da inserire nel vocabolario */

    for (i=0;i<=9;i++)

    {

        printf("\nParola %d ",i);

        gets(vocab[i]); /*3*/

    }/* fine ciclo for */
```

```
        /* Acquisisce la parola da cercare */
        printf("\n\nParola da cercare (Invio per finire) ");
        gets(parcerc);
        while (strcmp(parcerc,"")) /*4*/
        {
            /* Cerca la parola */
            trovata=0; /*5*/
            for (i=0;i<=9;i++)
            {
                if (!strcmp(parcerc,vocab[i]))  /*6*/
                {
                    trovata=1; /*7*/
                    break; /*8*/
                }/* fine if */
            }/* fine ciclo for */

            if (trovata) /*9*/
                printf("\nParola trovata");
            else
                printf("\nParola non trovata");

            /* Prossima parola da cercare */
            printf("\n\nParola da cercare (Invio per finire) ");
            gets(parcerc);
        }/* fine ciclo while */
    }/* fine programma */
```

La dichiarazione della riga `1` permette l'utilizzo all'interno del programma delle funzioni per il trattamento delle stringhe.

Nella riga `2` viene dichiarato l'array di caratteri `vocab[10][20]`. Il nostro vocabolario conterrà quindi un massimo di `10` parole ognuna composta di un massimo di `20` caratteri. La prima parola è contenuta in `vocab[0]` e con `vocab[2][7]`, qualora dovesse interessare, si individuerebbe l'*ottavo carattere* della *terza*

parola del vocabolario. Ogni riga della tabella conterrà una parola e in ogni colonna ci sono i caratteri che compongono le parole. L'array `parcerc[20]` servirà per contenere la parola da cercare.

Nella riga `3` si acquisiscono le parole da inserire nel vocabolario: la prima volta l'istruzione si leggerà `gets(vocab[0])`, quindi `gets(vocab[1])` e infine `gets(vocab[9])` in relazione ai diversi valori assunti dal contatore `i`. L'uso di un solo indice dipende dal fatto che, con la `gets`, si acquisisce una stringa e quindi nel nostro caso una intera riga della matrice. Se avessimo voluto acquisire le stringhe un carattere per volta, si sarebbero dovuti gestire due cicli uno annidato nell'altro, così come nell'esempio della tavola pitagorica, contenenti l'istruzione `scanf("%c",&vocab[i][j])`. Alla fine avremmo dovuto aggiungere il carattere *null*.

Nella riga `4` si gestisce la condizione di uscita dal ciclo. La stringa ricevuta dall'input viene comparata con la *stringa vuota* (due doppi apici consecutivi): se alla richiesta di input della stringa da cercare si risponde premendo solamente il tasto Invio, viene acquisita una stringa contenente solo il carattere terminatore *null*. Ciò rende falsa la condizione e quindi il ciclo ha termine (`strcmp` in questo caso restituisce `0`).

La variabile `trovata` che viene azzerata nella riga `5` registrerà l'eventuale ritrovamento della stringa all'interno del vocabolario. Il valore assegnato non ha il senso di un valore numerico a tutti gli effetti ma il senso di un valore logico (0 = falso, un valore non nullo=vero). Una variabile usata in questo modo viene chiamata solitamente *variabile logica*, *switch* o *flag*.

Nella `6` si confronta la parola da cercare via via con le parole contenute nel vocabolario. L'equivalenza delle due stringhe, quella da cercare e la parola del vocabolario esaminata, rende vera la condizione. In questo caso, riga `7`, si *rende vera* la variabile `trovata` assegnandole un valore non nullo e con la `break` della riga `8` si forza l'uscita dal ciclo poiché è inutile continuare a confrontare la parola cercata con il resto delle parole del vocabolario: la ricerca viene sospesa.

Nella riga `9` si esegue un test sul flag `trovata`: se è vero (cioè è stata eseguita l'istruzione della riga `7`) la parola è stata trovata, se è falso (è rimasto inalterato il valore assegnato nella istruzione della riga `5`) la ricerca ha avuto esito negativo.

Costruzione di un programma: lo sviluppo *top-down*

Accade spesso, specie nei problemi complessi, che una stessa sequenza di istruzioni compaia nella stessa forma in più parti dello stesso programma o che sia utilizzata in più programmi. Gli algoritmi riguardano elaborazioni astratte di dati che possono essere adattate a problemi di natura apparentemente diversi (dal punto di vista informatico due problemi sono diversi se *necessitano di elaborazioni diverse* e non se *trattano di cose diverse*). Per fare un esempio riguardante altre discipline basta pensare per esempio alla Geometria: il calcolo dell'area di una superficie varia in relazione alla forma geometrica diversa e non alla natura dell'oggetto. L'area di una banconota o di una lastra di marmo si calcolerà sempre allo stesso modo trattandosi in ambedue i casi di rettangoli, quindi per tornare al nostro caso, l'elaborazione riguardante il calcolo dell'area di un rettangolo ricorrerà nei problemi di calcolo di blocchi di marmo così come nei problemi di calcolo di fogli su cui stampare banconote.

Per risparmiare un inutile lavoro di riscrittura di parti di codice già esistenti, i linguaggi di programmazione prevedono l'uso dei ***sottoprogrammi***. Sostanzialmente un sottoprogramma è una parte del programma che svolge una funzione elementare.

L'uso di sottoprogrammi non è solo limitato al risparmio di lavoro della riscrittura di parti di codice, ma è anche uno strumento che permette di affrontare problemi complessi riconducendoli a un insieme di problemi di difficoltà via via inferiore. Tutto ciò consente al programmatore un controllo maggiore sul programma stesso *nascondendo* nella fase di risoluzione del singolo sottoprogramma, le altri parti in modo tale da *isolare* i singoli aspetti del problema da risolvere.

Si tratta del procedimento di stesura *per raffinamenti successivi* (o ***top-down***). Quando la complessità del problema da risolvere cresce, diventa difficoltoso tenere conto *contemporaneamente* di tutti gli aspetti coinvolti, fin nei minimi particolari, e prendere *contemporaneamente* tutte le decisioni realizzative: in tal caso sarà necessario procedere per approssimazioni successive, cioè decomporre il problema iniziale in sottoproblemi più semplici. In tal modo si affronterà la risoluzione del problema iniziale considerando in una prima approssimazione risolti, da altri programmi di livello gerarchico inferiore, gli aspetti di massima del problema stesso. Si affronterà quindi ciascuno dei sottoproblemi in modo analogo.

In definitiva si comincia specificando la sequenza delle fasi di lavoro necessarie anche se, in questa prima fase, possono mancare i dettagli realizzativi: si presuppone infatti che tali dettagli esistano già. Se poi si passa all'esame di una singola fase di lavoro, questa potrà ancora prevedere azioni complesse ma riguarderà, per come è stata derivata, una *parte* del problema iniziale. Iterando il procedimento mano a mano si prenderanno in esame programmi che riguardano parti sempre più limitate del problema iniziale. In tal modo la risoluzione di un problema complesso è stata ricondotta alla risoluzione di più problemi semplici (tanti quante sono le funzioni previste dalla elaborazione originaria).

Le tecniche di sviluppo per raffinamenti successivi suggeriscono cioè di scrivere subito il programma completo, come se il linguaggio di programmazione a disposizione fosse di livello molto elevato ed orientato proprio al problema in esame.

Tale programma conterrà, oltre alle solite strutture di controllo, anche operazioni complesse che dovranno poi essere ulteriormente specificate. Queste operazioni verranno poi descritte in termini di operazioni ancora più semplici, e così via fino ad arrivare alle operazioni elementari fornite dal linguaggio di programmazione utilizzato.

inizio

Un esempio di sviluppo *top-down*

Applichiamo il procedimento descritto alla risoluzione del seguente problema:

In una località geografica sono state rilevate ogni 2 ore le temperature di una giornata. Si vuole conoscere la temperatura media, l'escursione termica e lo scostamento medio dalla media.

Si tratta di scrivere un programma che richiede alcune elaborazioni statistiche su una serie di valori. Si ricorda che la media aritmetica di una serie *n* di valori è data dal rapporto fra la somma dei valori della serie e il numero *n* stesso. L'escursione termica è in pratica il campo di variazione cioè la differenza fra il valore massimo e il valore minimo della serie di valori. Lo scostamento medio è la media dei valori assoluti degli scostamenti dalla media aritmetica, dove lo scostamento è la differenza fra il valore considerato della serie e la media aritmetica.

Scriviamo il programma nel modo seguente:

```
Inizio

Acquisizione temperature rilevate

Calcolo media e ricerca massimo e minimo

Calcolo escursione termica

Calcolo scostamento medio

Comunicazione risultati

Fine
```

In questa prima approssimazione si sono evidenziati i risultati intermedi da conseguire affinché il problema possa essere risolto. Non si parla di istruzioni eseguibili ma di *stati di avanzamento* del processo di elaborazione: per il momento non c'è niente di preciso ma **il nostro problema è stato ricondotto a 5 problemi ognuno dei quali si occupa di una determinata elaborazione**. Viene messa in evidenza la sequenza delle operazioni da effettuare: l'escursione termica si può, per esempio, calcolare solo dopo la ricerca del massimo e del minimo.

Si noti che ad ogni fase di lavoro è assegnato un compito specifico ed è quindi più facile la ricerca di un eventuale sottoprogramma errato: se lo scostamento medio è errato e la media risulta corretta è chiaro che, con molta probabilità, l'errore è localizzato nel penultimo sottoprogramma.

Il primo sottoprogramma possiamo già tradurlo in istruzioni eseguibili. È opportuno tenere presente che a questo livello il problema da risolvere riguarda solamente l'acquisizione delle temperature rilevate. Il resto del programma, a questo livello, non esiste.

```
Acquisizione temperature

Inizio

Per indice da 0 a 11

Ricevi temperatura rilevata

Fine-per

Fine
```

Passando al dettaglio del secondo sottoprogramma possiamo pensarlo composto da una fase di inizializzazione dell'accumulatore della somma dei termini della serie e delle variabili che conterranno il massimo ed il minimo

della serie stessa. La seconda fase è il calcolo vero e proprio. Anche qui il problema è limitato solo alla parte specificata.

```
Calcolo media e ricerca massimo e minimo

Inizio

Inizializza Somma Valori

Considera primo elemento serie come Massimo e Minimo

Aggiorna Somma e cerca Massimo e Minimo

Calcola Media

Fine



Aggiorna Somma e cerca Massimo, Minimo

Inizio

Per indice da 1 a 11

Aggiorna Somma con elemento considerato

Se elemento<Minimo

Sostituisci elemento a Minimo

Altrimenti

Se elemento>Massimo

Sostituisci elemento a Massimo

Fine-se

Fine-se

Fine-per

Fine
```

Il terzo sottoprogramma è immediato:

```
Calcolo escursione termica

Inizio

Escursione = Massimo – Minimo

Fine
```

Il quarto sottoprogramma, come il secondo, prevede una inizializzazione:

```
Calcolo scostamento medio

Inizio
```

```
Azzera Somma scostamenti

Aggiorna Somma

Calcola Media scostamenti

Fine



Aggiorna Somma scostamenti

Inizio

Per indice da 0 a 11

Se elemento>Media aritmetica

Scostamento = elemento-Media

Altrimenti

Scostamento = Media-elemento

Fine-se

Aggiorna Somma scostamenti con scostamento

Fine-per

Fine
```

L'ultimo sottoprogramma è immediato:

```
Comunicazione risultati

Inizio

Comunica Media

Comunica Escursione termica

Comunica Scostamento medio

Fine
```

Il nostro programma a questo punto è interamente svolto. Per ogni sottoprogramma ci si è occupati di un solo aspetto dell'elaborazione: ciò rende la stesura del programma più semplice ed inoltre la manutenzione del programma stesso diventa più semplice. Ogni sottoprogramma diventa più semplice da controllare rispetto al programma nel suo complesso.

Il programma è stato diviso praticamente in 7 parti. Ognuna può essere trattata, in sede di codifica, come un sottoprogramma a parte come anche si può decidere di lasciare ai sottoprogrammi, vedi seconda e quarta fase, il compito dell'elaborazione e lasciare le inizializzazioni nel programma principale (il primo che si è scritto, quello cioè dove è evidenziata tutta l'elaborazione da svolgere). Si può anche decidere, sempre nella seconda e quarta fase per esempio, di considerare le inizializzazioni parte integrante della elaborazioni stessa e quindi rispettare la prima suddivisione di massima del programma, saltando la seconda suddivisione.

Il *processo di scomposizione successiva* non è fissato in maniera univoca: dipende fortemente dalla *soggettività* del programmatore. Non ci sono regole sulla *quantità di pezzi* in cui scomporre il programma. Ci sono delle

indicazioni di massima che suggeriscono di limitare il singolo segmento in maniera sufficiente a che il codice non superi di molto la pagina in modo da coprire, con lo sguardo, l'intero o quasi programma e limitare il singolo segmento a poche azioni di modo che sia più semplice isolare eventuali errori. Inoltre il sottoprogramma deve essere quanto più possibile isolato dal contesto in cui opera, cioè il sottoprogramma deve ***avere al suo interno tutto ciò di cui ha bisogno*** e non fare riferimento a dati particolari presenti nel programma principale. Ciò porta ad alcuni indubbi vantaggi:

- Il sottoprogramma è facilmente esportabile. Se dalla scomposizione di altri programmi si vede che si ha necessità di utlizzare elaborazioni uguali si può ricopiare il sottoprogramma. È evidente che affinché ciò sia possibile è necessario che il sottoprogramma non faccia riferimento a contesti che in questo caso potrebbero essere diversi. Se, inoltre, il sottoprogramma effettua una sola operazione si potrà avere più opportunità di inserirlo in nuove elaborazioni.
- La manutenzione del programma è semplificata dal fatto che, facendo il sottoprogramma una sola elaborazione e, avendo al suo interno tutto ciò che serve, se c'è un errore nella elaborazione questo è completamente isolato nel sottoprogramma stesso e, quindi, più facilmente rintracciabile.
- Qualora si avesse necessità di modificare una parte del programma, ciò può avvenire facilmente: basta sostituire solamente il sottoprogramma che esegue l'elaborazione da modificare. Il resto del programma non viene interessato dalla modifica effettuata.

L'utilizzo di sottoprogrammi già pronti per la costruzione di un nuovo programma porta ad una metodologia di sviluppo dei programmi che viene comunemente chiamata ***bottom-up*** poiché rappresenta un modo di procedere opposto a quello descritto fino ad ora. Si parte da sottoprogrammi già esistenti che vengono assemblati assieme a nuovi per costruire la nuova elaborazione. In definitiva *"... si può affermare che, nella costruzione di un nuovo algoritmo, è dominante il processo top-down, mentre nell'adattamento (a scopi diversi) di un programma già scritto, assume una maggiore importanza il metodo bottom-up."* (N.Wirth)

inizio

Comunicazioni fra sottoprogrammi

Seguendo il procedimento per scomposizioni successive si arriva alla fine ad un programma principale e ad una serie di sottoprogrammi. Il programma principale *chiama* in un certo ordine i sottoprogrammi; ogni sottoprogramma oltre che *chiamato* può anche essere il *chiamante* di un ulteriore sottoprogramma. Terminato il sottoprogramma l'esecuzione riprende, nel chiamante, dall'istruzione successiva alla chiamata.

Si può dire che tutti i sottoprogrammi fanno parte di un insieme organico: ognuno contribuisce, per la parte di propria competenza, ad una elaborazione finale che è quella fissata dal programma principale. Si tratta in definitiva di un sistema dove l'elaborazione finale richiesta è frutto della cooperazione delle singole parti; ogni sottoprogramma (unità del sistema) riceve i propri input (intesi come somma delle informazioni necessarie all'espletamento delle proprie funzioni) dai sottoprogrammi precedenti, e fornisce i propri output (intesi come somma delle informazioni prodotte al suo interno) ai sottoprogrammi successivi. Per questi ultimi, le informazioni suddette saranno gli input della propria parte di elaborazione.

Quanto detto porterebbe alla conclusione che tutti i sottoprogrammi, facendo parte di un insieme organico, lavorano sulle stesse variabili. D'altra parte se, per esempio, il reparto carrozzeria e il reparto verniciatura operano nella stessa fabbrica di automobili, è ovvio che il reparto verniciatura si occuperà delle stesse carrozzerie prodotte dall'altro reparto. Così in effetti è stato per esempio per i linguaggi di programmazione non strutturati: i sottoprogrammi condividevano le stesse variabili, il sottoprogramma eseguiva una parte limitata di elaborazione ma era fortemente legato al contesto generale. La portatilità di un sottoprogramma, in queste condizioni, è estremamente problematica richiedendo pesanti modifiche al codice stesso (si pensi al fatto che il programma destinazione non usa le stesse variabili del programma che ospitava originariamente il sottoprogramma o, peggio ancora, usa variabili con nomi uguali ma con significati diversi).

Per garantire quanto più possibile la portatilità e l'indipendenza dei sottoprogrammi, i linguaggi strutturati adottano un approccio diverso distinguendo le variabili in base alla ***visibilità*** (in inglese *scope*). In relazione alla visibilità le variabili si dividono in due famiglie principali:

- Variabili **globali** visibili cioè da tutti i sottoprogrammi. Tutti i sottoprogrammi possono utilizzarle e modificarle. Sono praticamente patrimonio comune.
- Variabili **locali** visibili solo dal sottoprogramma che li dichiara. Gli altri sottoprogrammi, anche se chiamati, non hanno accesso a tali variabili. La variabile locale è definita nel sottoprogramma ed è qui utilizzabile. Se viene chiamato un sottoprogramma le variabili del chiamante sono mascherate e riprenderanno ad essere visibili quando il chiamato terminerà e si tornerà al chiamante. L'ambiente del chiamante (l'insieme delle variabili con i rispettivi valori) a questo punto verrà ripristinato esattamente come era prima della chiamata.

Per quanto ribadito più volte sarebbe necessario utilizzare quanto meno possibile (al limite eliminare) le variabili globali per ridurre il più possibile la dipendenza dal contesto da parte del sottoprogramma.

Riguardando però l'elaborazione dati comuni, è necessario che il programma chiamante sia in condizioni di poter comunicare con il chiamato. Devono cioè esistere delle *convenzioni di chiamata* cioè delle convenzioni che permettono al chiamante di comunicare dei *parametri* che rappresenteranno gli input sui quali opererà il chiamato. D'altra parte il chiamato avrà necessità di tornare al chiamante dei parametri che conterranno i risultati della propria elaborazione e che potranno essere gestiti successivamente. Queste convenzioni sono generalmente conosciute come ***passaggio di parametri***.

Il passaggio di parametri può avvenire secondo due modalità:

- Si dice che un parametro è **passato per valore** (dal chiamante al chiamato) se il chiamante comunica al chiamato il valore che è contenuto in quel momento in una sua variabile. Il chiamato predisporrà una propria variabile locale che conterrà tale valore. Il chiamato può operare su tale valore, può anche modificarlo ma tali modifiche riguarderanno solo la copia locale su cui sta lavorando. Terminato il sottoprogramma la varabile locale scompare assieme al valore che contiene e viene ripristinata la variabile del chiamante con il valore che essa conteneva prima della chiamata al sottoprogramma.
- Si dice che un parametro è **passato per riferimento** o per indirizzo se il chiamante comunica al chiamato *l'indirizzo di memoria* di una determinata variabile. Il chiamato può utilizzare, per la variabile, un nome diverso ma le locazioni di memoria a cui ci si riferisce sono sempre le stesse. Viene semplicemente stabilito un riferimento diverso alle stesse posizioni di memoria: ogni modifica effettuata si ripercuoterà sulla variabile originaria anche se il nuovo nome cessa di esistere alla conclusione del sottoprogramma.

Per sintetizzare praticamente su cosa passare per valore e cosa per riferimento, si può affermare che gli input di un sottoprogramma sono passati per valore mentre gli output sono passati per riferimento. Gli input di un sottoprogramma sono utili allo stesso per compiere le proprie elaborazioni mentre gli output sono i prodotti della propria elaborazione che devono essere resi disponibili ai sottoprogrammi successivi.

inizio

Tipi di sottoprogrammi

Si è parlato di sottoprogrammi in modo generico perché si volevano evidenziare le proprietà comuni. In genere si fa distinzione fra due tipi di sottoprogrammi:

- Le **funzioni**. Sono sottoprogrammi che *restituiscono* al programma chiamante un valore. La chiamata ad una funzione produce al ritorno un valore che, per esempio, potrà essere assegnato ad una variabile. Le funzioni per quanto detto vengono utilizzate principalmente a destra del segno di assegnamento.
- Le **procedure**. Sono sottoprogrammi che *non restituiscono* alcun valore; si occupano di una fase della elaborazione

È opportuno osservare che quanto espresso prima non esaurisce le comunicazioni fra sottoprogrammi. Da quanto detto infatti potrebbe sembrare che tutte le comunicazioni fra chiamante e chiamato si esauriscano, nella migliore delle ipotesi (funzioni), in un unico valore. In realtà la comunicazione si gioca principalmente sul passaggio di parametri, quindi una procedura può modificare più variabili: basta che riceva per riferimento tali variabili.

Nel linguaggio C ogni sottoprogramma ha un nome e i sottoprogrammi vengono chiamati specificandone il nome. Vedremo in seguito le convenzioni che regolano il passaggio di parametri.

inizio

I puntatori: operatori *&* e *

Si è avuto modo di conoscere l'operatore `&` (*indirizzo di*) utilizzato nella `scanf` per indicare che l'input deve essere conservato all'indirizzo di memoria corrispondente alla variabile specificata. L'operatore `*` è il secondo operatore utilizzato per la manipolazione di indirizzi di memoria: è utilizzato per specificare un **puntatore**.

Un puntatore è una variabile che contiene un indirizzo di memoria. Un puntatore è quindi una variabile che non contiene dati ma l'indirizzo dei dati. Una variabile per poter essere utilizzata come puntatore è necessario che sia dichiarata come tale. Si consideri il frammento di programma successivo:

```
..

int *x, *y; /*1*/

int a,b,c; /*2*/

..

a = 10; /*3*/

c = a; /*4*/

x = &a; /*5*/

y = x; /*6*/

*x = 20; /*7*/

b = a; /*8*/

..
```

Nella `1` vengono dichiarati due *puntatori ad interi*. Con tale terminologia si vuole intendere che tali variabili (`x` e `y`) conterranno ognuna il primo indirizzo di una quantità di locazioni di memoria (`2` nel nostro caso) tali da poter contenere un dato di tipo specificato.

Nella `3`, come noto, si assegna il valore `10` alla variabile `a` dichiarata nella `2`.

Nella `5` si assegna ad `x` l'indirizzo di `a`. Si noti il fatto che, essendo stato `x` dichiarato puntatore, l'unica cosa che può ricevere correttamente è un indirizzo: sarebbe errore assegnare ad `x` il valore di `a`. Questa assegnazione, in pratica, crea un nuovo riferimento alla stessa zona di memoria (quella identificata dalla variabile `a`). Si noti la differenza con l'assegnazione effettuata nella `4`. L'assegnazione della `4` duplica in `c` l'attuale valore conservato in `a`: si può modificare il valore contenuto in `c` senza che ciò comporti alcunché per `a` (sono variabili diverse). Nella `5` invece `x` rappresenta un nuovo riferimento alla stessa zona di memoria:

modificando il contenuto della zona di memoria interessata da `x` si modifica il contenuto della zona di memoria associata al simbolo `a`.

L'assegnazione della `6` è pienamente legittima: `x` e `y` sono entrambi puntatori. Da questo momento anche `y` sarà un riferimento alla stessa area di memoria di `a`.

L'assegnazione della `7` si legge: *nella locazione puntata da `x` mettiamo il valore `20`*. In definitiva si sta variando il contenuto della variabile `a`. Si sottolinea la differenza di significato con la `6`. Ovviamente poteva essere assegnato, oltre che un valore costante, il valore contenuto in una variabile. Si noti che, affinchè sia possibile effettuare una simile assegnazione, è indispensabile che il puntatore sia stato inizializzato: nell'esempio tale operazione è effettuata nella `5`.

La `8` è una comune assegnazione solo che `b`, a differenza di `c` nella `4`, riceve il valore modificato di `a` (modifica effettuata in conseguenza di `5` e `7`).

Il linguaggio C fa un uso intensivo dei puntatori. Si tratta di uno strumento fra i più potenti ma anche delicati: un uso improprio di puntatori può portare a gravi guasti del sistema o a errori difficili da identificare. In ogni caso rappresenta uno dei punti di forza del linguaggio C ed è bene acquisire comprensione e padronanza di tale strumento.

inizio

I sottoprogrammi in C: le funzioni. Istruzione *return*

Nel linguaggio C la maggior parte di quello che si usa è costituito da funzioni. Per poter simulare le procedure che non ritornano alcun valore è stato introdotto il tipo `void`. Il tipo `void` o *tipo indefinito* è utilizzato dal linguaggio C tutte le volte che il valore di ritorno di una funzione non deve essere preso in considerazione. In pratica nel linguaggio C le procedure sono funzioni che restituiscono un `void`.

Possiamo sintetizzare in tre fasi la costruzione e l'uso di una funzione:

- La **definizione** della funzione cioè l'elenco delle operazioni svolte dalla funzione stessa. La definizione comincia specificando il tipo di valore ritornato, subito dopo viene specificato il nome scelto a piacere dal programmatore e seguente le regole della scelta del nome delle variabili, viene quindi specificato l'elenco dei parametri e infine le dichiarazioni locali e le istruzioni così come dallo schema seguente:

  ```
  tipo-ritornato nome-funzione(dichiarazione parametri)
  {
      dichiarazioni ed istruzioni
  }
  ```

  Le definizioni di funzioni possono essere scritte in qualsiasi punto del programma: verranno mandate in esecuzione in seguito alla chiamata e quindi non ha alcuna importanza il posto fisico dove sono allocate. Lo standard ANSI ha però fissato delle convenzioni secondo le quali le definizioni delle funzioni è bene codificarle dopo il `main`. D'altra parte il `main` stesso non è altro che una funzione particolare che viene eseguita per prima. Il programma quando viene eseguito effettua una chiamata alla funzione `main`. Il tipo-ritornato non è specificato perché si sottintende `int`: quando non è specificato il tipo, il linguaggio C assume per default il tipo `int` sebbene, per motivi di comprensibilità, sia sempre opportuno dichiarare il tipo. In definitiva all'atto dell'esecuzione il programma effettua una chiamata alla funzione `main`, questa può restituire al chiamante (in questo caso il Sistema Operativo) un valore intero. Tale valore potrebbe per esempio essere utilizzato per stabilire l'azione da svolgere in relazione ad una terminazione anomala del programma.

Fra le istruzioni contenute nella definizione della funzione particolare importanza assume l'istruzione `return` utilizzata per ritornare al chiamante un valore. La sintassi dell'istruzione prevede di specificare dopo la parola chiave `return` un valore costante o una variabile compatibile con il tipo-ritornato dalla funzione. Es.

```
return 5; /* Ritorna al chiamante il valore 5 */

return a; /* Ritorna al chiamante il valore contenuto nella variabile a */
```

Non è importante che le definizioni di tutte le funzioni usate in un programma seguano l'ordine con cui sono chiamate sebbene, per motivi di chiarezza e leggibilità, è opportuno che sia così e che si segua l'ordine specificato prima. In ogni caso la funzione con il nome `main` viene eseguita per prima in qualunque posto sia messa e le funzioni vengono eseguite nell'ordine in cui sono chiamate.

- Il **prototipo** della funzione. Si tratta in sintesi di ripetere all'inizio del programma, prima della definizione della funzione `main`, le dichiarazioni delle funzioni definite dopo. Si riscrive in pratica quanto specificato nella prima riga della definizione della funzione.

  I prototipi sono stati introdotti dallo standard ANSI per poter permettere al compilatore di effettuare un maggiore controllo sui tipi di parametri: conoscendoli in anticipo infatti, all'atto della chiamata che avviene a partire dalla funzione `main` (la prima eseguita), è possibile stabilire se i parametri passati sono congruenti con quelli attesi.

  Nella costruzione di programmi complessi capita di utilizzare molte funzioni. In questo caso le funzioni sono raggruppate in **librerie** e i rispettivi prototipi sono raggruppati nei files di intestazione (**header** files). Gli header sono file di tipo testo che hanno estensione "`.h`". Si è avuto modo di utilizzare librerie di funzioni fin dall'inizio. Per esempio sia `scanf` che `printf` sono funzioni contenute in una libreria di sistema che viene *inclusa*, all'atto della compilazione, nel nostro programma. I prototipi di tali funzioni sono nello header `stdio.h` che, per tali motivi, viene incluso all'inizio del programma.

- La **chiamata** della funzione. Consiste semplicemente nello specificare, laddove occorre utilizzare l'elaborazione fornita dalla funzione, il nome della funzione stessa accompagnato dall'elenco dei parametri passati. Il programma chiamante può utilizzare il valore restituito dalla funzione e in tal caso il nome della funzione figurerà, per esempio, in una espressione. Il chiamante può anche trascurare il valore restituito anche se non è di tipo `void`. Basta utlizzare la funzione senza assegnare il suo valore restituito. La funzione `scanf`, per esempio, restituisce un intero che indica quanti input sono stati effettuati, ma negli esempi fin qui trattati non si è mai utilizzato tale valore di ritorno. Ciò potrebbe per esempio essere utile per conoscere, prima di proseguire nell'elaborazione, se tutti gli input richiesti sono quelli che ci si aspettava:

  ```
  ..

  while(scanf("%d,%d",&a,&b)!=2);

  ..
  ```

  L'istruzione specificata è un ciclo che controlla se il valore di ritorno della `scanf` è diverso da 2 (non ci sono stati due input validi) e in tal caso ripete l'input (il ciclo ha solo l'istruzione di controllo).

inizio

Passaggio di parametri in C

Nel linguaggio C tutti i parametri sono passati alle funzioni **per valore** e ciò allo scopo di isolare quanto più possibile la funzione e renderla riutilizzabile. Per passare alla funzione un parametro per riferimento è

necessario ricorrere ai puntatori. L'unica eccezione a quanto detto riguarda **gli array** che **sono sempre passati per indirizzo**.

Per prendere confidenza con il passaggio di parametri ad una funzione e con la scelta di cosa passare per valore e cosa per riferimento, esaminiamo le chiamate di alcune funzioni comuni utilizzate fino ad ora:

- `scanf("%d",&a);`

  In questo caso sono passati alla funzione `scanf` due parametri: il valore di una stringa e l'indirizzo di una variabile. Il primo parametro è passato per valore, il secondo per riferimento. Lo scopo della funzione `scanf` è quello di acquisire valori dalla tastiera e depositarli in variabili di memoria. L'output della funzione è il valore assegnato alla variabile: la funzione ha necessità di conoscere che tipo di valore deve attendere (la stringa di formattazione è un suo input) e produce una variazione del valore attualmente contenuto in `a` (il valore contenuto in `a` è un suo output).

- `printf("Risultato %d",risult);`

  Lo scopo della funzione `printf` è quello di effettuare delle visualizzazioni su video: ha necessità di conoscere cosa visualizzare (i parametri sono passati per valore poiché costituiscono degli input per la funzione). La funzione non ha output: non produce risultati da utilizzare in altre funzioni, non c'è quindi passaggio di parametri per riferimento.

- `gets(frase);`

  Anche in questo caso, nonostante l'apparenza, c'è un parametro passato per riferimento. La `gets` è una funzione di input di stringhe e quindi `frase` sarà dichiarato come array di `char`. **Negli array il nome è un puntatore**: indica l'indirizzo del punto di inizio dell'array (praticamente la posizione dell'elemento di indice `0`). Alla funzione `gets` è passato un riferimento all'array `frase`.

  A questo punto dovrebbe essere chiara una osservazione effettuata durante l'illustrazione della funzione di trattamento stringhe `strcpy`. La necessità di tale funzione è dovuta al fatto che, se per esempio `s1` ed `s2` sono due stringhe, l'assegnazione solita `s1=s2`, per il significato che assumono in C i nomi degli array, farebbe in modo che `s1` diventi un nuovo riferimento all'unica stringa `s2` (`s1` ed `s2` sarebbero due puntatori allo stesso array). La funzione `strcpy(s1,s2)` crea invece in `s1` un duplicato di `s2`: ci saranno in questo caso due stringhe con lo stesso valore.

La funzione, nella sua definizione, preparerà variabili locali per poter ricevere i parametri passati dal chiamante. Nel caso di passaggio per indirizzo, per poterlo conservare, la funzione utilizzerà una variabile puntatore che è l'unica che può conservare un indirizzo. In questo ultimo caso, come specificato prima, ogni variazione effettuata dalla funzione sul contenuto della variabile locale si ripercuoterà sulla variabile corrispondente del chiamante.

I parametri passati dal chiamante vengono comunemente chiamati argomenti o **parametri reali**, mentre quelli presenti nella definizione della funzione vengono chiamati **parametri formali**.

inizio

Un esempio pratico: elaborazioni statistiche

A questo punto si sono acquisite tutte le informazioni che servono per codificare il programma delle elaborazioni statistiche di temperature. Al solito le righe significative sono accompagnate da commenti numerici utilizzati per le osservazioni successive:

```
/*
```

```
Temperatura media, escursione termica e scostamento medio
di una serie di temperature rilevate in una località
Le temperature sono state rilevate ogni 2 ore
*/
#include <stdio.h>
#define RILEVAZ 12 /*1*/

/* prototipi di funzione */
void rilevaTemp(int *t); /*2*/
void aggiorna(int *t,int *st,int *tmn,int *tmx); /*2*/
float scostMed(int *t,float mt); /*2*/

main()
{
    int temp[RILEVAZ],tempMax,tempMin,escur;
    float medTemp,medSco;
    int sommaTemp;

    /* Acquisizione temperature rilevate */
    rilevaTemp(temp); /*3*/

    /* Calcolo media e ricerca massimo e minimo */
    sommaTemp = tempMin = tempMax = temp[0];
    aggiorna(temp,&sommaTemp,&tempMin,&tempMax); /*4*/
    medTemp = (float) sommaTemp/RILEVAZ; /*5*/

    /* Calcolo escursione termica */
    escur = tempMax-tempMin;

    /* Scostamento medio */
    medSco = scostMed(temp,medTemp); /*6*/
```

```
    /* Comunicazione risultati */

    printf("\nMedia temperature rilevate -> %2.2f",medTemp);

    printf("\nEscursione termica ---------> %d",escur);

    printf("\nScostamento medio ----------> %2.2f",medSco);

}/* fine programma */


/*--------------------*/
/* SVILUPPO FUNZIONI */
/*--------------------*/

/* Lettura temperature */

void rilevaTemp(int *t) /*7*/

{

    int i;


    printf("\nAcquisizione temperature rilevate\n");

    for (i=0;i<RILEVAZ;i++) /*8*/

    {

        printf("Temperatura ore %d ",i*2);

        scanf("%d",&t[i]);

    }/* fine ciclo for */

}/* fine funzione rilevaTemp */



/* Aggiornamento Somma temperature, temp massima e minima */

void aggiorna(int *t,int *st,int *tmn,int *tmx) /*9*/

{

    int i;


    for (i=1;i<RILEVAZ;i++)

    {

        *st += t[i];
```

```
            t[i]<*tmn ? *tmn=t[i] : (t[i]>*tmx ? *tmx=t[i] : 0); /*10*/

        }/* fine ciclo for */

}/* fine funzione aggiorna */



/* Calcolo scostamento medio */

float scostMed(int *t,float mt) /*11*/

{

    float somScost,scost,media;

    int i;


    somScost = 0;

    for (i=0;i<RILEVAZ;i++)

    {

        scost = t[i]>mt ? t[i]-mt : mt-t[i];

        somScost += scost;

    }

    media = somScost/RILEVAZ;

    return media; /*12*/

}/* fine funzione scostMed*/
```

Nella riga `1` si dichiara una costante. Tale dichiarazione essendo esterna rispetto a tutte le funzioni ha visibilità globale: tutte le funzioni possono utilizzarla. Le variabili globali, se ci sono, devono essere dichiarate in questa posizione in modo da essere disponibili per tutte le funzioni.

Nelle `2` sono dichiarati i prototipi delle funzioni utilizzate. A questo punto quello che importa è *il tipo* di parametro. I nomi dei parametri, potendo mancare, sono aggiunti qui per chiarezza.

Nella riga `3` viene effettuata una chiamata alla funzione `rilevaTemp` passando il parametro `temp`. Si tratta in pratica di un puntatore all'array, infatti per quanto osservato precedentemente, il nome di un array rappresenta il puntatore al primo elemento dell'array stesso. In questo modo la `main` fa conoscere a `rilevaTemp` la posizione in memoria dell'array `temp`.

Nella `4` viene effettuata una chiamata alla funzione `aggiorna`. Tale funzione ha il compito di aggiornare i valori contenuti in `sommaTemp`, `tempMin` e `tempMax` (che quindi per la funzione rappresentano degli output). L'array `temp` per la funzione è un input (non viene effettuata da `aggiorna` alcuna modifica sull'array), basterebbe quindi un passaggio per valore del parametro solo che un array, per quanto osservato precedentemente, è sempre passato per indirizzo: possono essere passati per valore solo i singoli elementi di un array essendo questi delle variabili come tutte le altre.

Nella riga `5` si effettua una divisione fra interi, se si vuole conservare anche la parte decimale si deve effettuare un casting.

Nella riga `6` viene effettuata una chiamata alla funzione `scostMed`, solo che in questo caso a differenze delle precedenti chiamate alle altre funzioni (righe `3` e `4`), volendo utilizzare il valore ritornato dalla funzione, tale valore è assegnato ad una variabile.

Nella riga `7` viene definita la funzione `rilevaTemp` che restituisce un valore indefinito (`void`). La funzione prepara il puntatore `t` per conservare l'indirizzo che gli viene passato dal chiamante: `main` passa alla funzione il puntatore `temp`, la funzione conserva tale puntatore in `t` che è locale, così come la variabile intera `i` definita nella riga successiva. Quando la funzione termina scompaiono sia `t` che `i` solo che `t` era un riferimento a `temp` e, quindi, le modifiche effettuate a `t` nella funzione, praticamente erano modifiche effettuate a `temp`. La variabile `i` invece, al termine della funzione, scompare senza lasciare alcuna traccia della sua esistenza.

La condizione di controllo del ciclo della riga `8` è espressa come `i<RILEVAZ` perché gli elementi dell'array sono in mumero di `RILEVAZ` e quindi le posizioni utilizzabili vanno da `0` a `RILEVAZ-1`.

Per quanto riguarda la definizione della funzione della riga `9` valgono considerazioni analoghe alla definizione della funzione della riga `7` per quanto riguarda l'array. Per gli altri parametri vengono preparati, dalla funzione, dei puntatori per poter permettere alla stessa la modifica dei valori contenuti.

Nella riga `10` è codificata utilizzando l'operatore `?` una doppia condizione nidificata. Se la prima condizione è vera (`t[i]<*tmn`) viene effettuata l'assegnazione specificata. Se risulta falsa viene testata una nuova condizione (`t[i]>*tmx`). Anche qui se tale condizione è vera viene effettuata una assegnazione, se è falsa non c'è alcuna elaborazione: viene assegnato il valore `0` che, non essendoci alcuna variabile pronta per riceverlo, viene trascurato. Il valore deve essere espresso poiché l'operatore `?` deve restituire sempre un valore.

Nella riga `11` viene definita una funzione che ritorna un valore di tipo `float`. Altrove questi sono gli unici sottoprogrammi che vengono chiamati funzioni, laddove i precedenti due (che in C sono funzioni che restituiscono un valore di tipo `void`) verrebbero chiamati procedure. Anche in questo caso nelle due righe successive sono dichiarate delle variabili locali.

Nella riga `12` viene ritornato al chiamante il valore atteso. La variabile `media` è dichiarata di tipo `float` in concordanza con il tipo di ritorno della funzione. La variabile è locale e quindi cessa la sua esistenza al cessare della funzione, ma in questo modo si rende noto al chiamante il valore contenuto in essa.

inizio

Due osservazioni. Il qualificatore *const*

Sono necessarie due osservazioni sulla codifica effettuata:

La trasposizione in codifica delle singole fasi dell'elaborazione è stata effettuata con una combinazione di commenti e funzioni. Le funzioni avrebbero potuto essere di più: per esempio tante quante sono le parti in cui si è scomposto il programma. La scelta effettuata in questo caso ha tenuto conto del fatto di evitare una estrema polverizzazione in funzioni che avrebbe reso più difficoltosa la comprensione dell'algoritmo utilizzato per la risoluzione del problema. Quando si trattava di poche istruzioni, queste sono state inglobate nel programma principale. È bene ripetere che questa o altra scelta che è possibile effettuare, dipende dal modo come il programmatore intende organizzare le diverse fasi di lavoro. Bisogna in ogni caso ricordare che ogni parte del programma deve esprimere chiaramente quello che fa; organizzare il programma in modo che sia chiaro *cosa si fa e dove* ed inoltre ogni fase di lavoro deve quanto più possibile essere isolata.

L'utilizzo della costante globale `RILEVAZ` rende dipendenti le funzioni da tale costante rendendo più problematico il riutilizzo delle funzioni stesse e il rintracciamento di eventuali errori.

È possibile dichiarare le costanti anche all'interno di una funzione utilizzando la parola chiave `const`. Es.

Programma **A**	Programma **B**

`..`
`#define RILEVAZ 12`
`..`
`main()`
`{`
`..`
	`..`
	`main()`
	`{`
	`const int RILEVAZ = 12;`
	`..`

Quella espressa nel Programma **A** non è una vera e propria istruzione ma una **direttiva al preprocessore**. Prima della compilazione viene effettuata una *sostituzione*: tutte le ricorrenze di `RILEVAZ` sono sostituite col valore `12` e quindi viene effettuata la compilazione. `RILEVAZ`, praticamente, nel programma e in memoria non esiste. Le direttive al preprocessore sono azioni da svolgere prima di passare alla compilazione del programma. Anche `#include <stdio.h>` è una direttiva al preprocessore: essa ordina di includere nel programma lo header `stdio.h` prima di procedere alla compilazione.

Nel Programma **B** viene invece dichiarata una variabile di tipo `int` che contiene il valore `12`. Il qualificatore `const` avvisa che il contenuto della variabile associata non può essere modificato.

Se si utilizza la via suggerita dal Programma **B**, le funzioni non possono conoscere la quantità degli elementi presenti nell'arrray (la dichiarazione è locale): è necessario prevedere un ulteriore parametro che consenta di poter passare alle funzioni tale informazione. Le funzioni che elaborano l'array conterranno, nella definizione, un nuovo parametro che consenta di ricevere il valore della quantità di elementi presenti nell'array stesso.

L'uso della costante globale se da una parte rende le funzioni dipendenti da tale dichiarazione, dall'altra permette di semplificare il passaggio di parametri tra funzioni (si ha necessità di un parametro in meno).

Un uso interessante del qualificatore `const` consiste nell'eliminazione di una limitazione più volte messa in evidenza: il passaggio, necessariamente per riferimento, di un array alle funzioni. Basta specificare `const` per avvisare che l'array non deve essere modificato. In pratica l'array viene passato per valore.

Es.

```
void rilevaTemp(int *t);

void aggiorna(const int *t,int *st,int *tmn,int *tmx);

float scostMed(const int *t,float mt);
```

Alla prima funzione viene passato il riferimento all'array. Il compito di tale funzione è infatti quello di riempire l'array con i dati provenienti dall'input.

Le funzioni `aggiorna` e `scostMed` non modificano l'array, è quindi giusto che se ne impedisca una modifica anche accidentale: l'array in questi casi è un input.

Nonostante si passi alla funzione un puntatore, quindi un indirizzo di memoria, la `const` inpedisce la modifica del valore puntato.

Le strutture

Una struttura è un insieme di variabili di uno o più tipi, raggruppate da un nome in comune. Anche gli array sono collezioni di variabili come le strutture solo che un array può contenere solo variabili dello stesso tipo, mentre le variabili contenute in una struttura non devono essere necessariamente dello stesso tipo.

Le strutture del linguaggio C coincidono con quelli che in Informatica sono comunemente definiti **record**. Il raggruppamento sotto un nome comune permette di rappresentare, tramite le strutture, *entità* logiche in cui le variabili comprese nella struttura rappresentano gli *attributi* di tali entità.

Per esempio con una struttura possiamo rappresentare l'entità `libro` i cui attributi potrebbero essere: `titolo`, `autore`, `editore`, `prezzo`. In tale caso la definizione potrebbe essere:

```
struct libro
{
    char titolo[50];
    char autore[20];
    char editore[20];
    long int prezzo;
};
```

La sintassi del linguaggio prevede, dopo la parola chiave `struct`, un nome che identificherà la struttura (il *tag* della struttura). Racchiuse tra le parentesi vengono dichiarate le variabili che fanno parte della struttura (i *membri* della struttura). È bene chiarire che in questo modo si definisce la struttura logica `libro`, che descrive l'aspetto della struttura, e non un posto fisico dove conservare i dati. In pratica si può considerare come se si fosse definito, per esempio, come è composto il tipo `int`: ciò è necessario per poter dichiarare variabili di tipo `int`.

In C è possibile dichiarare la struttura assieme alle variabili, qui si preferisce tenere distinte le due cose per ragioni di organizzazione dei programmi come si vede nel frammento di programma successivo.

```
#include <stdio.h>
...
struct libro  /*1*/
{
    char titolo[50];
    char autore[20];
    char editore[20];
    long int prezzo;
};
```

```
...

main()

{

    struct libro lib1,lib2; /*2*/

    ...

    gets(lib1.titolo); /*3*/

    ...

    printf("%ld",lib2.prezzo); /*4*/

    ...

}
```

Nella riga `1` si dichiara il tipo `struct libro`. La dichiarazione è globale in modo che tutte le funzioni presenti possano accedere a tale definizione. Si evita così di ridefinire la struttura in ogni funzione.

Nella `2` si dichiarano due *istanze* (`lib1` e `lib2`) di `libro` (due variabili di tipo `struct libro`).

La funzione della `3` effettua l'input nella componente `titolo` della istanza `lib1`. Per fare riferimento ad una componente deve infatti essere specificata l'istanza oltre che la componente: i due nomi sono separati dal punto. La specifica dell'istanza è indispensabile: ci sono infatti, nel caso in esame, due `titolo` (quello di `lib1` e quello di `lib2`). Analogo discorso vale per la componente `prezzo`: la `4` infatti si occupa dell'output di tale componente relativamente all'istanza `lib2`.

inizio

Le tabelle: array di strutture

La struttura array consente di trattare insiemi di dati omogenei solo che, i dati rappresentati in un array, possono possedere un solo attributo. Per conservare dati individuati da più attributi occorre ricorrere alle tabelle.

Una **tabella** è un insieme finito di elementi ciascuno costituito da una *chiave* (informazione che serve per individuare l'elemento) e da un *valore* (informazioni associate alla chiave). Si può pensare la tabella come ad un quadro con le seguenti caratteristiche:

1. Un numero di riga (chiave) che è associato alla riga e quindi la individua in maniera univoca
2. Un numero finito di righe in ognuna della quale è conservato un elemento dell'insieme di dati rappresentato (il valore associato alla chiave)
3. Un numero finito di colonne in ognuna delle quali è rappresentato un attributo dell'elemento conservato nella riga.

Per esempio potremmo conservare in una tabella i libri utilizzati in un anno scolastico. In ogni riga della tabella sono conservate le informazioni relative ad un libro. Ogni colonna della tabella conterrà un attributo del libro (per es. nella prima colonna ci sarà il titolo, nella seconda l'autore ecc..).

Nel linguaggio C una tabella è rappresentata tramite un array di strutture. Risolviamo, come esempio di applicazione degli array di strutture, il seguente problema:

Acquisiti i dati riguardanti i libri acquistati in un anno scolastico, si vogliono conoscere i dati dei libri che hanno avuto un costo superiore ad un input dato.

Dal punto di vista informatico è un problema di selezione: data una tabella si tratta di riempire una nuova tabella con le copie delle righe della prima tabella che soddisfano ad una certa condizione

```
/*
Dato un elenco di libri e un prezzo dato in input
fornisce dati sui libri che costano di più
*/
#include <stdio.h>
#define QLIB 5 /* Quantità libri caricati */
/* Implementazione delle strutture dati utilizzate */
struct libro  /*1*/
{
    char titolo[50];
    char autore[20];
    char editore[20];
    long int prezzo;
};

struct libro libreria[QLIB],libcost[QLIB]; /*2*/
int qlc=-1; /*3*/

/* Funzioni per l'elaborazione delle strutture dati */
void carica();
void ricerca(long int pz);
void comunica();

main()
{
    long int prezinp;
    /* Caricamento libri e acquisizione prezzo */
    printf("\n\nComunica titolo libri con");
```

```
    printf("\ncosto maggiore di un input\n");
    carica();
    for(;;) /*4*/
    {
        printf("\nPrezzo da confrontare (0 per finire)= ");
        scanf("%ld",&prezinp);

        if(!prezinp) /*5*/
            break;

        /* Ricerca libri con costo maggiore */
        qlc=-1; /*6*/
        ricerca(prezinp);

        /* Output titolo libri con costo maggiore */
        printf("\nLibri con prezzo maggiore di %ld\n",prezinp);
        comunica();
    }/* fine for */

    return 0;
}/* fine programma */


/* Caricamento libri */
void carica()
{
    int i;
    printf("\nInserimento di %d libri",QLIB);
    for(i=0;i<QLIB;i++)
    {
        fflush(stdin); /*7*/
```

```
        printf("\nLibro n°%d\n",i+1);
        printf("Titolo : ");
        gets(libreria[i].titolo); /*8*/
        printf("Autore : ");
        gets(libreria[i].autore); /*8*/
        printf("Editore : ");
        gets(libreria[i].editore); /*8*/
        printf("Prezzo : ");
        scanf("%ld",&libreria[i].prezzo); /*8*/
    }/* fine for */
}/* finr funzione carica */



/* ricerca libri con costo maggiore */
void ricerca(long int pz)
{
    int i;

    for(i=0;i<QLIB;i++)
    {
        if(libreria[i].prezzo>pz)
        libcost[++qlc]=libreria[i]; /*9*/
    }/* fine for */
}/* fine funzione ricerca */



/* Libri con prezzo maggiore */
void comunica()
{
```

```
        int i;
        for(i=0;i<=qlc;i++)
        {
            puts(libcost[i].titolo); /*10*/
            puts(libcost[i].autore); /*10*/
            puts(libcost[i].editore); /*10*/
            printf("%ld\n",libcost[i].prezzo); /*10*/
        }/* fine for */
    }/* fine funzione comunica */
```

A cominciare dalla 1 sono implementate le strutture elaborate nel programma. Nella riga 1 si dichiara la struttura libro. Tale dichiarazione, come la dichiarazione della costante QLIB, è globale: tutte le funzioni conoscono tale struttura.

Nella 2 sono dichiarati due array di libro: libreria e libcost (due tabelle). Anche in questo caso si tratta di dichiarazioni globali: tutte le funzioni di manipolazione delle strutture possono accedere a tali variabili.

La variabile intera qlc dichiarata in 3 viene utilizzata per indicare la posizione dell'ultimo elemento inserito nella tabella di output. Attualmente la tabella non contiene righe e quindi tale variabile è inizializzata al valore specificato. L'assenza di una analoga variabile per la tabella di input dipende dal fatto che si ipotizza che la quantità di libri caricati nella tabella di input sia sempre QLIB: non è quindi necessario utilizzare una ulteriore variabile.

Dalla 4 inizia un ciclo senza fine (non sono previste condizioni per l'uscita dal ciclo: nella for ci sono solo istruzioni vuote). L'uscita dal ciclo è determinata dal verificarsi dall'*evento* specificato nella 5: l'input nullo nella variabile prezinp.

L'assegnazione della 6 è necessaria per azzerare la tabella di output e prepararla per contenere i risultati di una nuova elaborazione. Il programma infatti permette di specificare più volte il prezzo da cercare e per ogni prezzo fornisce l'elenco dei libri con le caratteristiche richieste.

La funzione fflush() utilizzata nella 7 serve per pulire il buffer associato ad un flusso di dati da eventuali caratteri residui da operazioni precedenti. Il nome simbolico stdin indica lo standard input cioè la tastiera. In definitiva si svuota la zona di memoria associata alla tastiera in modo da preparla per i nuovi dati in ingresso.

Nelle 8 vengono effettuati gli input delle componenti della riga i-esima della tabella libreria.

Nelle 9 in pratica si copia la riga i della tabella libreria nella riga qlc della tabella libcost, copiandone le corrispondenti componenti. L'utilizzo di strutture permette, con un'unica operazione di copia sul nome della struttura, di copiare tutte le componenti.

La funzione comunica restituisce in output la riga i-esima, con le sue componenti, nelle 10.

inizio

Strutture dei dati: generalità

Negli esempi che si sono esaminati si sono incontrati, oltre che dati elementari come, per esempio, quelli di tipo `int`, `float` o `char`, si sono esaminati due tipi di **aggregati di dati**: gli **array** e le **strutture**.

In questa sede ci si propone di esporre altri tipi di aggregati o, come si dice più correttamente altri tipi di strutture di dati. Una trattazione completa delle strutture dei dati va oltre gli scopi dei presenti appunti: qui si vogliono fornire i concetti generali, le operazioni fondamentali e le prime strutture dati.

Per prima cosa occorre distinguere tra:

- **Struttura astratta**: identifica il modo come i dati sono collegati logicamente fra di loro. Con questo termine si fa riferimento alle leggi che definiscono le relazioni fra i dati.
- **Struttura interna**: identifica il modo come i dati sono rappresentati nella memoria di un elaboratore. Per rappresentare una struttura astratta in memoria si dovrà tenere conto delle leggi che raggruppano i dati e del modo come è possibile tradurre tale struttura astratta in una struttura interna senza stravolgere le proprietà di aggregazione dei dati: cioè adattare la struttura astratta alle esigenze di conservazione in memoria dei dati (*allocazione* in memoria della struttura astratta).

Le strutture interne sono sostanzialmente riconducibili a due tipi:

1. **Struttura sequenziale**: è costituita da un *insieme definito* di elementi, ognuno costituito da una o più celle di memoria, che sono *fisicamente adiacenti* in memoria. La caratteristica di essere la struttura composta da un numero definito di elementi, la rende adatta a conservare strutture astratte di tipo *statico*. Strutture cioè che restano pressocché uguali nel tempo: se si alloca spazio per `n` elementi, la quantità degli elementi effettivamente presenti in memoria sarà uguale o quasi uguale a tale numero. Il fatto poi che la lunghezza degli elementi sia la stessa per tutti porta ad una facilità di accesso ad un elemento qualsiasi della struttura. Se infatti si vuole conoscere l'indirizzo dell'`i`-esimo elemento basta calcolare: $IND(x_i)=IND_b+(i-1)l$ dove con $IND(x_i)$ si è indicato l'indirizzo da trovare (l'elemento è x_i), IND_b è l'*indirizzo base* (l'indirizzo del primo elemento della struttura), i è la posizione relativa dell'elemento cercato ed l è la lunghezza comune. Il modo come è organizzata tale struttura se da una parte permette di rintracciare facilmente e in maniera rapida, qualsiasi elemento, dall'altra si presenta in maniera rigida: non solo il numero degli elementi non può essere variato (essendo gli elementi adiacenti è necessario predisporre una serie di locazioni di memoria consecutive nelle quali conservare la struttura), ma anche l'inserimento o l'eliminazione di un nuovo elemento fra due già esistenti risultano operazioni se non impossibili, quanto meno estramemente dispendiose (occorre infatti per esempio nel caso di inserimento fra l'elemento x_i e l'elemento x_{i+1}, se è possibile se cioè la struttura non è interamente occupata, spostare gli elementi dalla posizione $i+1$ in poi in modo da creare lo spazio per l'elemento da inserire e quindi procedere con l'inserimento).
2. **Struttura concatenata**: è costituita da un insieme di elementi disposti in posizioni qualsiasi della memoria. Ogni elemento è costituito da una parte nella quale è contenuta l'informazione da conservare e un indicatore (puntatore) che fornisce l'indirizzo dell'elemento che deve intendersi successivo logicamente al corrente. L'ultimo elemento contiene, nel puntatore, un valore particolare (*puntatore nullo*) che permette di riconoscere l'elemento come l'ultimo della catena. Poiché non è possibile calcolare direttamente l'indirizzo di un elemento generico della struttura, per accedere ad esso è necessario, conoscendo l'indirizzo del primo elemento, percorrere sequenzialmente la catena finché non si trova l'elemento cercato. In questa struttura non è necessario conoscere in anticipo la quantità degli elementi che ne dovranno far parte: quando occorre inserire un nuovo elemento basta avere a disposizione una quantità di memoria tale da poter conservare l'elemento. L'aggancio dell'elemento alla struttura è facilitato dall'uso dei puntatori: il prossimo elemento non è quello che si trova conservato nelle locazioni successive rispetto all'elemento considerato (come nella struttura sequenziale), ma quello che segue logicamente cioè quello indicato dal puntatore. L'inserimento dell'elemento E_k fra l'elemento E_i e l'elemento E_{i+1}, per esempio, è una operazione che può essere compiuta in modo semplice tenendo presente che ogni elemento, per esempio E_k, è formato dall'informazione I_k e dal puntatore P_k. Per effettuare l'inserimento basta sistemare E_k in una zona di memoria disponibile, per esempio all'indirizzo $IND(E_k)$, mettere nel puntatore P_k dell'elemento E_k il valore attualmente contenuto in P_i (in modo che E_{i+1} segua E_k), mettere nel puntatore E_i il valore $IND(E_k)$ in modo che

E_k risulti il successivo di E_i. La struttura si presta, per le sue caratteristiche, a rappresentare strutture atratte di tipo *dinamico*; strutture cioè che cambiano frequentemente nel tempo. Un elemento della struttura potrà avere anche un puntatore al precedente e, in questo caso, si parlerà di **catena bidirezionale**, così come si potrà avere l'ultimo elemento che punta al primo e, in questo caso, si parlerà di **catena circolare**.

Per riassumere brevemente si può dire che la struttura sequenziale presenta il vantaggio di fornire accesso immediato a qualsiasi elemento della struttura a costo di una rigidità della struttura (numero elementi da conoscere all'atto dell'allocazione, difficoltà di aggiornamento della struttura). La struttura concatenata, al contrario, può conservare elementi finché c'è spazio in memoria, le operazioni di inserimento o cancellazione di elementi vengono svolte in maniera agevole, ma risulta impossibile l'accesso diretto ad un elemento qualunque della struttura.

inizio

Strutture astratte

Gli **array**, le prime e fondamentali strutture astratte, sono già state introdotte e si è inoltre parlato della loro implementazione in linguaggio C, così come sono state trattate le **tabelle**. Altre due strutture di cui si vedranno in seguito le implementazioni e le operazioni più comuni sono le **pile** e le **code**. Di **grafi** e **alberi** si accennerà solo alle definizioni fondamentali.

La **pila o stack** è una lista di elementi (insieme di elementi ordinati) in cui tutti gli inserimenti o le estrazioni di nuovi elementi avvengono ad un estremo della lista (la *cima*). La pila viene anche spesso indicata, a causa della gestione degli elementi, come **lista LIFO** (**L**ast **I**n **F**irst **O**ut cioè l'ultimo inserito sarà il primo estratto). In pratica in una pila sono definite due operazioni: l'inserimento in cima di un nuovo elemento (*push*) e l'estrazione sempre dalla cima di un elemento (*pop*).

La **coda** è una lista di elementi in cui gli inserimenti avvengono dopo l'ultimo elemento (*fondo* della coda) e le estrazioni avvengono dall'estremo opposto (*testa* della coda). La coda viene anche indicata come **lista FIFO** (**F**irst **I**n **F**irst **O**ut cioè il primo inserito sarà anche il primo estratto).

Il **grafo** è costituito da un insieme di *nodi* (le informazioni) e *archi* (collegamenti fra le informazioni). Ogni nodo può essere collegato a più altri nodi stabilendo così più modi di collegare logicamente i dati rappresentati nella struttura. Per esempio se in ogni nodo sono rappresentati i dati riguardanti un alunno di un istituto scolastico, un arco potrebbe portare al prossimo alunno della stessa classe frequentata e un altro arco potrebbe portare al prossimo alunno che risiede nella stessa città dell'alunno preso in esame. In questo modo un *cammino* (successione di nodi collegati) porta alla conoscenza di tutti gli alunni di una stessa classe e un secondo cammino porta all'elenco degli alunni residenti in un determinato paese. Se gli archi sono orientati (cioè se, essendo collegati i nodi n_i ed n_{i+1}, si può andare da n_i ad n_{i+1} ma non viceversa), il grafo si dice *orientato*.

L'**albero** è un grafo orientato che presenta le seguenti caratteristiche: esiste un nodo a cui non arrivano ma da cui si dipartono archi (*radice*), ogni nodo ha un solo arco che arriva ad esso ma può avere più archi che partono da esso, esistono nodi (*foglie* o nodi terminali) a cui porta un arco ma da cui non partono archi. Gli alberi vengono utilizzati per rappresentare strutture gerarchiche; nella pratica, anche al di fuori delle applicazioni informatiche, si trovano spesso applicazioni di tale struttura (alberi genealogici, famiglie zoologiche).

inizio

Array e aritmetica dei puntatori

Si è già fatto notare che, nel linguaggio C, il nome di un array è praticamente il puntatore al primo elemento della struttura ossia, per quanto detto precedentemente, l'indirizzo base. Ciò porta alla considerazione che l'elemento generico del vettore è accessibile oltre che, come già osservato, cioè come l'elemento di posizione `i`

della struttura, anche come l'elemento il cui indirizzo può essere calcolato a partire dall'indirizzo base, dalla lunghezza dell'elemento e dalla posizione relativa sebbene, per motivi legati alla comprensibilità delle istruzioni, è più opportuno accedere agli elementi di un array utillizzando l'indice.

Programma **A**

```
#define QEL 5...

void carica(int *v1)
{
   int i;
   for(i=0;i<QEL;i++)
   {
      printf("\nElemento n°%d ",i+1);

      scanf("%d",(v1+i));

   }/* fine for */

}

...void comunica(const int *v1)
   {
      int i;
      for(i=0;i<QEL;i++)
         printf("%d ",*(v1+i));
   }
```

Programma **B**

```
#define QEL 5...

void carica(int *v1)
{
   int i;
   for(i=0;i<QEL;i++)
   {
       printf("\nElemento n°%d",
               i+1);

       scanf("%d",&v1[i]);

   }/* fine for */

}

...void comunica(const int *v1)
   {
       int i;
       for(i=0;i<QEL;i++)
           printf("%d ",v1[i]);
   }
```

L'input e l'output, dell'elemento generico di un array di elementi di tipo `int`, nel Programma **B** viene effettuato al solito modo. Nel Programma **A** l'input viene effettuato specificando l'indirizzo dove deve essere conservato il dato. Il nome `v1` è un puntatore ed è bene chiarire che sommare il valore di `i` a tale puntatore significa, come fatto rilevare quando si è trattata la struttura sequenziale, spostarsi all'`i`-esimo elemento. In altri termini, per esempio: `a+1` se `a` è una variabile di tipo `int` vuol dire sommare il valore `1` al valore esistente, se invece `a` è un puntatore a `int` significa incrementare l'indirizzo contenuto in `a` di tante posizioni quante ne sono previste dal tipo `int`.Nel caso dell'istruzione di input l'istruzione `v1+i` equivale a IND_b`+(i-1)l` infatti `v1` rappresenta l'indirizzo base, `i-1` nel nostro caso si traduce in `i` poiché si parte dal valore `0`, in quanto a `l` questo è contenuto implicitamente nella dichiarazione del puntatore (è dichiarato come puntatore ad `int`, quindi il compilatore conosce la lunghezza degli elementi della struttura). In questo modo l'array viene gestito nella sua rappresentazione come struttura interna.

Nell'input non è utilizzato l'operatore `&` poiché in ciò che è scritto è già espresso un indirizzo. Un discorso simile occorre farlo per l'output: `v1+i` è un puntatore e quindi al programma interessa il valore contenuto nella memoria puntata.

Sui puntatori possono essere quindi effettuate operazioni aritmetiche, è questo infatti quello a cui si fa riferimento quando si parla di aritmetica dei puntattori, solo che bisogna tenere presente il senso che assumono tali operazioni: aggiungere il valore `1` a un puntatore vuol dire spostarsi di tante locazioni di memoria quante ne bastano per trovare il prossimo elemento dello stesso tipo. Da questo punto di vista dovrebbe risultare chiara, per esempio, la convenzione utilizzata nel passaggio di un array a più dimensioni ad una funzione: vengono infatti specificate tutte le dimensioni tranne la prima. Tutto ciò è comprensibile se si tiene presente che un array, per esempio a due dimensioni, è conservato in memoria per righe successive (la prima riga, poi la seconda ecc..); in questo caso la conoscenza della quantità di colonne serve per acquisire conoscenza sulle dimensioni del singolo elemento (in questo caso un elemento è composto da una intera riga dell'array).

inizio

Gestione di una pila

La pila può essere implementata in memoria utilizzando sia una struttura sequenziale se, per esempio, si prevede una dimensione massima che potrà avere la struttura, sia una struttura concatenata. Nel programma seguente viene utilizzata una struttura sequenziale per gestire una pila di libri. In questo programma vengono messe in evidenza principalmente le funzioni per la gestione di una pila.

```
/*

Gestione di uno STACK con un elenco di libri

*/

#include <stdio.h>

/* Implementazione dello Stack */

#define DIMSTACK 8 /*1*/

struct libro

{

    char titolo[50];

    char autore[20];

    char editore[20];

    long int prezzo;

};

struct libro libreria[DIMSTACK];

int cima=-1;

/* Funzioni per la gestione dello Stack */

int pieno(); /*2*/

void push(const struct libro *lib); /*3*/

int vuoto(); /*2*/

struct libro pop(); /*3*/

/* Altre funzioni del programma */

void inserisci();

void estrai();

main()

{

    int scelta;
```

```
    for(;;)
    {
        /* Menu operazioni disponibili */
        printf("\nGestione di una pila di libri\n"); /*4*/
        printf("\n1) Inserimento di un libro");
        printf("\n2) Estrazione di un libro");
        printf("\n0) Fine elaborazione\n");
        printf("\nScelta operazione (1,2,0) ");
        scanf("%d",&scelta);
        if(!scelta) /*5*/
            break;

        /* Richiama funzione scelta */
        switch(scelta)
        {
        case 1: inserisci();
        break;

        case 2: estrai();
        }/* fine switch */
    }/* fine for */

    return 0;
}/* fine programma */

/* Inserimento di un libro nella pila */
void inserisci()
{
    struct libro buflib; /*6*/
    if(pieno())
```

```
    { /*7*/
        printf("\n\aPresenti %d elementi",DIMSTACK);
        printf("\nPila piena: inserimento impossibile");
    }/* fine if */
    else
    {
        printf("\nInserimento di un libro"); /*8*/
        fflush(stdin);
        printf("\nTitolo : ");
        gets(buflib.titolo);
        printf("Autore : ");
        gets(buflib.autore);
        printf("Editore : ");
        gets(buflib.editore);
        printf("Prezzo : ");
        scanf("%ld",&buflib.prezzo);
        push(&buflib); /*9*/
    }/* fine else */
}/* fine funzione */



/* Estrazione di un libro */
void estrai()
{
    struct libro buflib;
    if(vuoto()) /*10*/
        printf("\n\aPila vuota: estrazione impossibile");
    else
    {
        buflib=pop(); /*11*/
```

```
        puts(buflib.titolo);
        puts(buflib.autore);
        puts(buflib.editore);
        printf("%ld\n",buflib.prezzo);
        fflush(stdin);
        while(!getchar()=='\n'); /*12*/
    }/* fine else */
}/* fine funzione */



/* Funzioni standard per la gestione di uno stack:
funzione che ritorna lo stato dello stack
*/
int pieno() /*13*/
{
    if(cima==DIMSTACK-1)
        return 1;
    else
        return 0;
}



/* Inserimento di un elemento nella pila */
void push(const struct libro *lib) /*14*/
{
    libreria[++cima]=*lib;
}
```

```
/* Funzione che ritorna lo stato dello stack */

int vuoto() /*15*/

{

    if(cima==-1)

        return 1;

    else

        return 0;

}
```

```
/* Prelievo di un elemento dalla pila */

struct libro pop() /*16*/

{

    return libreria[cima--];

}
```

A cominciare dalla 1 viene definita la struttura: la dimensione massima dello stack, la struttura dell'elemento generico, la struttura sequenziale su cui verrà implementata la pila. Il puntatore cima, attualmente inizializzato al valore −1, sarà l'indice utilizzato per rintracciare la posizione dell'elemento su cui effettuare le operazioni. Si ricorda che, in una pila, le operazioni vengono effettuate sempre ad un estremo.

Nelle righe 2 e 3 sono indicate le funzioni utilizzate per la manipolazione della struttura. Le funzioni possiamo raggrupparle in due categorie: le 2 restituiscono informazioni sulla struttura, le 3 forniscono le operazioni ammesse sulla struttura. Qui finisce la definizione della struttura fornita in termini di tipo di dati che ne fanno parte e di funzioni per la loro manipolazione.

Il programma presentato offre un menu di operazioni possibili (visualizzato a cominciare da 4) da cui si esce, tramite 5, digitando un opportuno valore. Il programma consente solo operazioni di inserimento ed estrazione di un libro dalla pila.

La funzione di inserimento prepara nella 6 un buffer su cui conservare temporaneamente il libro da inserire nella struttura. Prima di effettuare l'inserimento nella pila è necessario assicurarsi che ci sia posto: ciò viene fatto, effettuando nella 7 una chiamata alla funzione pieno, affinché si sappia se è possibile effettuare l'inserimento. L'inserimento effettivo viene svolto ricevendo, cominciando da 8, i dati forniti in input nella memoria temporanea buflib e comunicando nella 9 alla funzione push l'indirizzo di tale area di memoria.

La funzione di estrazione di un elemento dalla pila è simile alla funzione di inserimento. Si interroga la pila al fine di conoscere se ci sono elementi da prelevare (linea 10). Se ci sono elementi da prelevare, si riceve dalla funzione pop, chiamata in 11, l'elemento cercato che viene conservato in un buffer appositamente predisposto.

La funzione `getchar` chiamata nella `12` serve per prelevare un carattere dal buffer di tastiera. Qui, dopo aver pulito il buffer di tastiera, si aspetta che arrivi da tastiera un carattere di *newline*, utilizzando un ciclo che preleva caratteri finché non si trova quello desiderato.

La funzione `pieno`, definita nella `13`, ritorna al chiamante un indicatore sullo stato della pila: se `cima` contiene l'indice dell'ultimo elemento la pila è piena.

La funzione `push`, definita in `14`, dopo aver aggiornato `cima` copia nello stack l'elemento attualmente conservato in una memoria temporanea. Dal modo come è definita tale funzione si può dedurre, come d'altra parte riportato nel programma, che l'operazione di inserimento non controlla se effettivamente c'è posto nella struttura. La funzione, per poter funzionare correttamente, è necessario che sia chiamata condizionatamente ad una risposta ottenuta dal richiamo della funzione `pieno`. La funzione riceve come parametro passato un puntatore al buffer che contiene l'elemento da inserire. Tale parametro poteva essere anche una variabile di tipo `libro`. La scelta, nel caso di strutture, di passare un puntatore al posto di una variabile è comune nei programmi in C: passando un puntatore si effettua una elaborazione più veloce (non si spreca tempo nell'effettuare una copia della struttura passata nel parametro. Si consideri che una struttura può contenere parecchi membri) ed inoltre non si occupa memoria per conservare la struttura in una variabile locale.

La funzione `vuoto`, definita in `15`, restituisce indicazioni sulla presenza di elementi nello stack. Per considerazioni analoghe a quelle fatte precedentemente, è necessario chiamarla prima di effettuare l'inserimento.

La funzione `pop` della `16` ha un funzionamento analogo alla `push`, solo che qui le operazioni hanno verso opposto: dalla pila al buffer.

inizio

Gestione di una coda

La gestione di una coda presenta dei punti in comuni con la gestione della pila, le funzioni che verranno presentate avranno dei punti in comune con le corrispondenti della pila.

Rispetto alla gestione della pila, la coda presenta principalmente una differenza sostanziale che comporta delle soluzioni implementative differenti rispetto a quelle usate nel primo caso.

- Una pila cresce solo da un lato e quindi basta, utilizzando una struttura sequenziale, tenere conto solo di un puntatore (`cima`) all'ultimo elemento inserito: se l'indicatore raggiunge in valore la dimensione della struttura sequenziale, la pila è piena.
- Una coda ha necessità di utilizzare due puntatori (`testa` e `fondo`): uno per i prelievi e uno per gli inserimenti. Anche in questo caso si utilizzerà una struttura sequenziale per implementare la coda solo che, stavolta, il puntatore `fondo` può raggiungere il valore della dimensione della struttura sequenziale, senza che tale condizione comporti come conseguenza il riempimento della coda. La struttura che si sta esaminando è di tipo dinamico, e quindi i prelievi e gli inserimenti si susseguono: può darsi che siano già stati prelevati, per esempio, tutti gli elementi meno uno. In tale caso la zona di memoria riservata alla struttura è praticamente vuota.

Per i motivi espressi prima è opportuno, in questo caso, utilizzare una *struttura circolare*. Se la struttura sequenziale prevede `n` elementi si conviene di adottare le seguenti convenzioni:

- Per $0 \leq i < (n-1)$ il successivo dell'elemento di posizione i sarà l'elemento di posizione $i+1$
- Per $i = n-1$ il succesivo sarà l'elemento di posizione 0. Praticamente arrivati all'ultimo elemento si ricomincia dal primo.

Sarà inoltre opportuno fare in modo che `testa` punti una posizione prima del primo elemento e che `fondo` punti all'ultimo inserito. Così facendo se da una parte si sarà costretti a non utilizzare un elemento della struttura (quello a cui punta `testa`), dall'altra la condizione `testa == fondo` sarà indicativa del fatto che la coda è vuota.

In seguito si esaminerà la nuova struttura con le sue caratteristiche. Il resto del programma può essere identico a quello visto prima per la gestione dello stack.

```
/* Implementazione della Coda */

#define DIMCODA 8 /*1*/

struct libro

{

    char titolo[50];

    char autore[20];

    char editore[20];

    long int prezzo;

};

struct libro libreria[DIMCODA];

int testa=-1;

int fondo=-1;

/* Funzioni per la gestione della Coda */

int piena(); /*2*/

void aggiungi(const struct libro *lib);

int vuota();

struct libro elimina();

/* Funzioni standard per la gestione di una coda:

funzione che ritorna informazioni sullo stato della coda

*/

int piena()

{

    int numel;

    numel=(fondo>=testa)?(fondo-testa):(fondo+DIMCODA-testa); /*3*/

    if(numel==DIMCODA-1) /*4*/

        return 1;

    else
```

```
        return 0;
}



/* Aggiunge un elemento alla coda */
void aggiungi(const struct libro *lib)
{
    fondo = ++fondo % DIMCODA; /*5*/
    libreria[fondo]=*lib; /*6*/
}



/* Funzione che ritorna informazioni sullo stato della coda */
int vuota()
{
    if(testa==fondo) /*7*/
        return 1;
    else
        return 0;
}



/* Elimina un elemento dalla coda */
libro elimina()
{
    testa = ++testa % DIMCODA; /*8*/
    return libreria[testa]; /*9*/
}
```

A cominciare da `1` è definita la struttura che è molto simile alla pila tranne per il fatto che qui ci sono due puntatori. Le funzioni di gestione, il cui prototipo si trova in `2`, sono anche esse formalmente uguali a quelle di gestione dello stack: cambieranno solo le operazioni di aggiornamento dei puntatori.

La funzione `piena` calcola in `3` il numero di elementi contenuti come differenza dei valori contenuti nei due puntatori della struttura. Poiché la struttura è circolare potrebbe essere che il valore di `fondo` sia minore del valore contenuto in `testa`: in tal caso bisogna effettuare una correzione su tale valore. In ogni caso `fondo` deve essere inteso maggiore di `testa`. In `4` viene controllato se la coda è piena; è già stato fatto osservare che una posizione di memoria resta inutilizzata.

Nella `5` si calcola il valore di `fondo` in relazione al fatto che si sta utilizzando una struttura circolare. Viene effettuata una operazione in *modulo* (l'operatore `%` è appunto l'operatore che restituisce il resto della divisione intera fra gli operandi): prima viene effettuato l'aggiornamento di `fondo` ad indicare che c'è un nuovo elemento, poi si calcola il resto in modo tale che il risultato sia sempre un numero compreso fra `0` e `DIMCODA-1`. L'elemento, attualmente conservato in un deposito temporaneo, viene poi copiato, in `6`, nella posizione individuata da `fondo`.

Come osservato prima la `7` fornisce la condizione per sapere se la coda è vuota.

L'elemento da eliminare si trova, come messo in evidenza in `9`, nella posizione `testa` che è calcolata in `8` sempre tenendo conto che si tratta di una struttura circolare.

Allocazione dinamica della memoria

Non sempre è nota a priori la dimensione di una struttura e quindi non è possibile utilizzare una struttura sequenziale che richiede la conoscenza di tale dimensione per poter allocare spazio in memoria. In questi casi viene utilizzata la struttura concatenata: si alloca spazio in memoria quando serve e tutto quello che serve, compatibilmente ovviamente con le risorse disponibili, e si collegano gli elementi fra di loro tramite puntatori, in modo che ogni elemento fornisca informazioni su dove trovare in memoria il successivo della lista.

Ora si esamineranno gli strumenti che mette a disposizione il linguaggio C per l'allocazione dinamica della memoria, in seguito si vedranno le funzioni principali per la gestione di una struttura concatenata.

sizeof

La funzione restituisce il numero di byte che occorrono per conservare un dato del tipo di quello specificato come argomento della funzione. Come argomento della funzione si può specificare un tipo o anche una espressione:

```
...
struct libro
{
    char titolo[50];
    char autore[20];
    char editore[20];
    long int prezzo;
};
...
int nbyte1,nbyte2;
...
nbyte1 = sizeof(int);
nbyte2 = sizeof(libro);
...
```

In questo caso la variabile `nbyte1` conterrà il numero di byte occorrenti per conservare in memoria un dato di tipo `int` e `nbyte2` il numero di byte occorrenti per conservare in memoria un dato di tipo `libro`.

Le altre due funzioni specificate di seguito richiedono l'inclusione, nel programma che intende utilizzarle, di un nuovo header. Occorre specificare: `#include <stdlib.h>`.

malloc

La funzione alloca una zona di memoria atta a contenere un dato tipo e restituisce un puntatore a tale zona di memoria. Come argomento la funzione richiede di conoscere il numero di byte da allocare.

Qualora non fosse disponibile la quantità di memoria richiesta, la funzione restituisce nel puntatore il valore `NULL` (*puntatore nullo*: costante simbolica definita in `stdlib.h` utilizzata come valore da attribuire ad un puntatore di fine lista o come inizializzazione di un puntatore).

```
...

struct libro

{

    char titolo[50];

    char autore[20];

    char editore[20];

    long int prezzo;

};

...

libro *p1; /*1*/

...

p1 = (libro *) malloc(sizeof(libro)); /*2*/

...
```

In `1` è dichiarato un puntatore alla struttura `libro`.

In `2` si richiede di allocare una zona di memoria tale da contenere una struttura di tipo `libro`; mediante un casting, il puntatore ritornato viene trasformato in puntatore alla struttura `libro`.

free

La funzione rilascia una zona di memoria prima allocata, per esempio, con una chiamata alla funzione `malloc`. La funzione richiede di conoscere, come parametro, il puntatore alla zona di memoria che si vuol liberare.

È opportuno notare la necessità dell'utilizzo di tale funzione. La memoria centrale è una risorsa limitata e quindi non è opportuno tenere allocata, per elementi che non sono più di nostro interesse, memoria che può essere utilizzata per conservare dati ancora utili per l'elaborazione da svolgere.

La chiamata: `free(p1);` libera la memoria allocata nell'esempio precedente.

inizio

Puntatori a strutture

Ricorre spesso, specie nelle elaborazioni che riguardano strutture dati dinamiche, così come si vedrà in seguito, la necessità di usare puntatori a strutture: per esempio per passare ad una funzione un riferimento ad una struttura.

Tali necessità sono così frequenti che il linguaggio C mette a disposizione uno speciale operatore: l'operatore −> (il trattino seguito dal simbolo di maggiore):

```
...
struct libro
{ /*1*/
    char titolo[50];
    char autore[20];
    char editore[20];
    long int prezzo;
};
...
void leggi(libro *pl); /*2*/
...
main()
{
    libro lib1;
    ...
    leggi(&lib1); /*3*/
    ...
}
void leggi(libro *pl)
{
    ...
    gets(pl->titolo); /*4*/
    gets(pl->autore); /*4*/
    ...
}
```

Nella `1` si dichiara la struttura che ha visibilità globale.

La `2` mostra il prototipo di una funzione che prepara un puntatore ad un dato di tipo `struct libro`.

Nella riga `3` si chiama la funzione `leggi` passando l'indirizzo di `lib1`, istanza della struttura `libro`.

Nelle 4 viene utilizzato l'operatore -> per accedere alle componenti della struttura. Una scrittura alternativa più lunga e, nel caso di strutture, poco usata sarebbe stata:

```
...

gets((*pl).titolo);

gets((*pl).autore);

...
```

inizio

Strutture con allocazione dinamica della memoria

Si è già trattato di pile e code implementandole in strutture sequenziali di memoria. È opportuno ricordare, come già fatto osservare a suo tempo, che la struttura sequenziale può andare bene solo a determinate condizioni:

- Se gli elementi della struttura sono pochi e può essere effettuata una previsione sulla quantità degli elementi che troveranno posto nella struttura, ed inoltre se in ogni momento si può prevedere una occupazione media della struttura stessa (si ricordi che la struttura occupa una porzione fissa di memoria e non può essere ampliata: è quindi indispensabile un calcolo abbastanza esatto dello spazio necessario per evitare sprechi ed avere sempre a disposizione posto per inserire elementi nella struttura)
- Se il problema da risolvere prevede l'uso di una sola struttura. Quando sono necessarie più strutture i problemi esposti in precedenza diventano più pressanti

L'allocazione dinamica permette di riservare spazio in memoria solo quando serve e solo quello che serve ed inoltre, l'uso di puntatori permette di operare più velocemente senza spostare fisicamente i dati, per esempio, nel passaggio di parametri. Sono questi due punti importanti nell'ottica di ottimizzazione di risorse: si tenga presente che l'elaboratore trova impiego nel trattamento di grosse masse di dati e quindi la razionalizzazione dello spazio occupato e la velocità di elaborazione sono due punti critici da curare particolarmente.

L'input di un nuovo elemento da inserire in una struttura di dati potrebbe essere svolto, utilizzando l'allocazione dinamica della memoria, secondo lo schema esposto.

```
/*

Esempio di funzione per l'input di una struttura

effettuato utilizzando l'allocazione dinamica della memoria

*/

#include <stdlib.h> /*1*/

struct libro

{

    char titolo[50];

    char autore[20];

    char editore[20];

    long int prezzo;
```

```
};

...

void inserisci()

{

    libro *buflib; /*2*/

    buflib=(libro *) malloc(sizeof(libro)); /*3*/

    if(buflib==NULL){ /*4*/

        printf("\n\aImpossibile allocare nuovo spazio");

    }

    else

    {

        printf("\nInserimento di un libro");

        fflush(stdin);

        printf("\nTitolo : ");

        gets(buflib->titolo); /*5*/

        printf("Autore : ");

        gets(buflib->autore); /*5*/

        printf("Editore : ");

        gets(buflib->editore); /*5*/

        printf("Prezzo : ");

        scanf("%ld",&buflib->prezzo); /*5*/

        ... /*6*/

    }/* fine else */

}/* fine funzione inserisci */
```

L'inclusione specificata in `1` permette di utilizzare le funzioni per l'allocazione dinamica della memoria.

Nella `2` si definisce un puntatore alla struttura `libro`. In questo modo non si alloca spazio in memoria come sarebbe se fosse stata definita una variabile di tipo `libro`. Lo spazio, necessario prima di effettuare l'input, viene allocato in `3`.

La funzione `malloc` quando non riesce ad allocare spazio ritorna il valore `NULL` nel puntatore: tale condizione è testata in `4`. Per il modo con il quale il C tratta le condizioni, la `4` poteva più propriamente essere espressa come: `if(!buflib)`.

Nelle `5` vengono effettuati gli input nelle varie componenti della struttura. Si noti l'uso dell'operatore `->` per referenziare le singole componenti.

A cominciare da `6` si può usare l'input effettuato per le elaborazioni richieste. Qualora richiesto, per passare l'input effettuato ad una funzione, basta passare il puntatore (si può, per esempio, utilizzare `const` se si tratta di passaggio per valore).

inizio

Stack con allocazione dinamica

Si riportano di seguito le funzioni per la gestione di uno stack solo che, stavolta, gli elementi nello stack sono allocati dinamicamente.

```
/* Funzioni per la gestione di un elenco di libri con uno STACK dinamico*/

#include <stdlib.h> /*1*/

/* Implementazione della struttura */

typedef struct rec *lpointer; /*2*/

typedef struct rec

{

    char titolo[50];

    char autore[20];

    char editore[20];

    long int prezzo;

    lpointer next; /*3*/

}libro;

lpointer cima=NULL; /*4*/

...

/* Funzioni per la gestione dello Stack */

void push(lpointer lib)

{ /*5*/

    lib->next=cima; /*6*/

    cima=lib; /*7*/

}




lpointer pop()

{ /*8*/
```

```
        lpointer lib;

        lib=cima; /*9*/

        if (cima) /*10*/

            cima=cima->next;

        return lib;

    }
```

L'inclusione specificata in 1 permette, all'interno del programma, di utilizzare le funzioni per l'allocazione dinamica della memoria.

Nella 2 la dichiarazione di un puntatore alla struttura `rec` è preceduta dalla dichiarazione di *definizione di tipo* `typedef`. Tale dichiarazione permette di utilizzare `lpointer` allo stesso modo dei tipi definiti dal C, come per esempio il tipo `int`. `lpointer` diventa, con questa dichiarazione, un *sinonimo* di `struct rec *`, così come `libro` diventa sinonimo di `struct rec`. Ciò permette di incrementare la leggibilità delle dichiarazioni, la quale cosa sarà di utilità specie nelle dichiarazioni complesse.

Così come messo in evidenza nelle righe precedenti, nella struttura `libro` oltre agli attributi visti negli altri esempi, viene dichiarato in 3 un puntatore al prossimo elemento della lista: `next` è una variabile di tipo `lpointer` cioè, come evidenziato in 2, un puntatore a `libro`. Come è possibile notare, la praticità di tale dichiarazione permette di abbreviare e rendere più chiara la natura della variabile `next`. Tale variabile viene utilizzata per collegare ogni elemento al suo successivo.

La dichiarazione presente in 4 è l'unica che alloca effettivamente spazio in memoria. Viene dichiarata la variabile `cima` che è di tipo `lpointer` e che viene inizializzata al valore `NULL`. Ciò indica che, in questo momento, la struttura è vuota.

Alla funzione `push` nella 5 viene passato un puntatore alla zona di memoria dove è conservato l'elemento da inserire nella struttura. L'attuale contenuto di `cima` viene copiato, nella 6, nel puntatore `next`. Se c'era `NULL` vorrà dire che questo elemento inserito sarà considerato l'ultimo, se invece puntava ad un elemento E_i, con tale assegnazione E_i diventa il successivo dell'elemento che si sta inserendo. Il puntatore `cima`, per la 7, da questo momento punterà all'ultimo elemento inserito.

La funzione `pop`, come definito in 8, ritorna un puntatore all'elemento estratto. Nella 9 viene definita una variabile di tipo `lpointer` alla quale viene assegnato il valore di `cima`. Come osservato, nei commenti riguardanti la `push`, `cima` punta all'ultimo elemento inserito. Nel caso in cui lo stack fosse vuoto, il valore assegnato sarebbe `NULL` per l'inizializzazzione della 4. Il programma chiamante deve controllare tale eventualità.

La funzione `pop`, come specificato in 8, ritorna un puntatore all'elemento estratto dallo stack. Il puntatore ritornato, come osservato precedentemente, assume, come messo in evidenza da 9, il valore di `cima`: è questo infatti il puntatore all'ultimo elemento inserito. Se lo stack è vuoto il puntatore, per la 4, varrà `NULL`. L'istruzione presente in 10 testa quest'ultimo caso: se c'era almeno un elemento, `cima` dovrà puntare al successivo.

Il chiamante dovrà verificare se la `pop` ritorna un puntatore valido e, finito l'utilizzo dell'elemento estratto, dovrà occuparsi di liberare, con una chiamata alla funzione `free`, la memoria occupata dall'elemento.

inizio

Coda con allocazione dinamica

La logica della gestione di una coda, fatte salve le specificità, è simile alla logica di gestione dello stack: si dovrà tenere conto solo del fatto che qui si devono gestire due puntatori.

```
/* Funzioni per la gestione di un elenco di libri con una CODA dinamica*/

#include <stdlib.h>

/* Implementazione della struttura */

typedef struct rec *lpointer;

typedef struct rec

{

    char titolo[50];

    char autore[20];

    char editore[20];

    long int prezzo;

    lpointer next;

}libro;

lpointer testa=NULL; /*1*/

lpointer fondo=NULL;

/* Funzioni per la gestione della Coda */

void aggiungi(lpointer lib)

{ /*2*/

    lib->next=NULL; /*3*/

    if (fondo) /*4*/

        fondo->next=lib;

    fondo=lib;

    if(!testa) /*5*/

        testa=lib;

}

lpointer elimina()

{ /*6*/

    lpointer lib;

    lib=testa; /*7*/
```

```
        if (testa) /*8*/
            testa=testa->next;
        if (!testa) /*9*/
            fondo=NULL;
        return lib;
    }
```

Le dichiarazioni presenti in `1` sono formalmente simili a quelle presenti nella gestione dello stack, eccezione fatta per la presenza, in questo caso, di due puntatori invece che uno solo: `testa` utilizzato per i prelievi, `fondo` per gli inserimenti.

La funzione `aggiungi` in `2`, si occupa dell'inserimento di un nuovo elemento nella coda. La funzione riceve un puntatore all'elemento da inserire e, per prima cosa (per la `3`) considera tale elemento ricevuto come ultimo e quindi registra nel suo puntatore il codice di fine coda. Si occupa poi di modificare il puntatore dell'ultimo elemento che era presente nella coda, ammesso che esista (condizione testata in `4`) e quindi di aggiornare `fondo` al nuovo elemento. Il controllo presente in `5` si occupa di verificare se `testa` era `NULL` (non c'erano cioè elementi prima di questo inserito) e, in tal caso fa in modo che tale puntatore conservi l'indirizzo dell'elemento inserito.

La funzione `elimina` della `6` preleva un elemento dalla coda, restituendo un puntatore allo stesso. Poiché `testa` punta al primo elemento da prelevare, è il valore di tale puntatore che deve essere restituito (istruzione `7`). Se, all'atto del prelievo, c'era almeno un elemento, `testa` dovrà puntare al prossimo: è di ciò che si occupa il controllo `8`. Il controllo presente in `9` si occupa di verificare se l'elemento prelevato era l'ultimo e, in tal caso reinizializza il puntatore `fondo`.

Così come osservato per la gestione dello stack, il chiamante, finito l'utilizzo dell'elemento estratto dalla coda, dovrà liberare la memoria occupata.

inizio

La ricorsione

Il linguaggio C consente l'uso di funzioni ricorsive. Una funzione ricorsiva è una funzione che richiama sé stessa (*ricorsione diretta*) o richiama una funzione che a sua volta la richiama (*ricorsione indiretta*). Affinché il procedimento abbia fine è necessario che siano verificate le due seguenti proprietà:

1. Debbono esistere dei parametri (*valori base*) per cui la funzione non richiami sé stessa
2. Ogni volta che la funzione richiama sé stessa i parametri devono essere più vicini ai valori base

L'uso di tali funzioni è di utilità per la codifica di algoritmi che sono definiti ricorsivamente. Un esempio classico di algoritmo ricorsivo è quello che definisce il fattoriale di un numero.

Il fattoriale del numero `n` intero positivo (cioè `n!`) è definito:

- uguale a `n*(n-1)!` se $n \neq 0$
- uguale a `1` se $n = 0$

In altri termini è vera la seguente uguaglianza: `n! = n*(n-1)*(n-2)*..2*1`. La prima definizione data è di tipo ricorsivo: il fattoriale di un numero viene calcolato come prodotto del numero stesso per il fattoriale del numero che lo precede. Affinché il procedimento sia finito si assume in pratica la posizione: `0! = 1`.

```
/* Fattoriale di un numero */

#include <stdio.h>

int calcFatt(int numero); /*1*/

main()

{

    int n,fat;

    printf("\nCalcola il fattoriale di un numero");

    printf("\n\nIntrodurre il numero ");

    scanf("%d",&n);

    fat=calcFatt(n); /*2*/

    printf("\n Fattoriale di %d = %d",n,fat);

}

/* Calcola Fattoriale utilizzando la Ricorsione*/

int calcFatt(int numero)

{

    int f;

    if (!numero) /*3*/

        f=1;

    else

        f=numero*calcFatt(numero-1); /*4*/

    return f;

}
```

Nella `1` viene definito il prototipo della funzione che calcola il fattoriale, funzione chiamata dal `main` nella riga `2`. A tale funzione è passato per valore il numero di cui si vuole calcolare il fattoriale.

La struttura condizionale che inizia da `3`, come si nota chiaramente, non fa altro che esprimere utilizzando le caratteristiche del linguaggio di programmazione, la definizione di fattoriale. La funzione chiama sé stessa (nella `4`) passando come parametro il valore, decrementato di una unità, del parametro ricevuto e in questo modo ci si approssima, come richiesto dalle proprietà espresse precedentemente, al valore `0` (valore base). Tale ultimo valore, in dipendenza della verità della condizione espressa in `3`, blocca il processo ricorsivo. La funzione `calcFatt` poteva, facendo riferimento alla seconda definizione data, essere scritta utilizzando una struttura ciclica:

```
/* Calcola Fattoriale utilizzando una Struttura Ciclica*/

int calcFatt(int numero)

{

    int i,f;

    f=1;

    for(i=numero;i>0;i--)

        f *= i;

    return f;

}
```

Nella prima versione di `calcFatt` il ciclo è in pratica sostituito dalla chiamata ricorsiva; l'algoritmo risulta più leggibile, più vicino al nostro modo di concepire la risoluzione del problema.

La ricorsione può essere utilizzata per scrivere qualsiasi funzione che utilizzi assegnazioni, istruzioni `if-else` e `while`. A titolo di esempio si propone il programma per il calcolo della somma di una serie di numeri, terminanti con il valore nullo, scritto utilizzando una funzione ricorsiva al posto di una struttura ciclica:

```
/* Somma una serie di numeri terminanti con 0 utilizzando una funzione
ricorsiva*/

#include <stdio.h>

int prossimo(void);

main()

{

    int somma;

    somma = prossimo();

    printf("\nSomma = %d",somma);

}

/* Funzione ricorsiva per il calcolo della somma */

int prossimo(void)

{

    int questo,somm;

    printf("\nNumero da Sommare =");

    scanf("%d",&questo);

    if (questo)

    { /*1*/

        somm = questo+prossimo();
```

```
        return somm;
    }
    else
        return 0;
}
```

Anche in questo caso, nella definizione della funzione, esiste un valore base (`questo==0`) e la funzione, richiamando sé stessa si approssima a tale valore (se si immaginano tutti gli input in una coda di attesa, ogni volta che se ne preleva uno con la funzione `scanf`, ci si avvicina alla fine della coda).

L'uso della funzione ricorsiva in questo caso rende più evidente l'algoritmo utilizzato. La struttura che comincia dalla riga `1` si può infatti interpretare:

> se questo numero preso in esame esiste (ha cioè un valore non nullo) somma questo numero con il prossimo
>
> altrimenti
>
> concludi la somma
>
> fine-se

inizio

Gestione di una lista concatenata

La lista concatenata è la struttura ideale da utilizzare quando si debbano rappresentare insiemi di dati in cui le operazioni di inserimento e cancellazione siano prioritarie. Tali operazioni, come si vedrà, vengono permesse in maniera estremamente agevole dalla struttura.

implementazione

Nel caso di una lista non c'è una allocazione preventiva della struttura: si definiscono le strutture di cui si ha bisogno, la memoria verrà allocata quando serve, quando cioè si tratterà di conservare gli elementi.

```
/*
Gestione di una LISTA con un elenco di libri
*/
#include <stdlib.h>
/* Implementazione della Lista */
typedef struct rec *lpointer;
typedef struct rec
{
    char titolo[50];
    char autore[20];
```

```
    char editore[20];

    long int prezzo;

    lpointer next;

}libro;

lpointer entryp=NULL; /*1*/
```

Le dichiarazioni presenti sono simili a quelle utilizzate nell'implementazione di uno stack: anche in questo caso si deve collegare ogni elemento con il suo successivo.

La dichiarazione presente in `1` è l'unica che alloca effettivamente spazio in memoria. Viene dichiarata la variabile `entryp` che è di tipo `lpointer` e che viene inizializzata al valore `NULL`. Questa rappresenta il puntatore al primo elemento della lista (*entry point*); i successivi elementi della lista saranno raggiungibili attraverso i puntatori contenuti in ogni nodo.

attraversamento

Definita la struttura concatenata, ci si occuperà delle operazioni fondamentali sulle liste. La prima operazione esaminata riguarda l'attraversamento di una lista: l'esame cioè di tutti i nodi presenti nella lista. Nel programma scritto di seguito vengono riportate solo le istruzioni riguardanti l'elaborazione richiesta; per la definizione della struttura si fa riferimento a quanto scritto in precedenza.

```
...

/* Funzioni per la gestione della Lista */

void esamina(lpointer p1); /*1*/

...

esamina(entryp); /*2*/

...

/* Funzioni per la gestione di una lista */

void esamina(lpointer p1)

{

    if(p1)

    { /*3*/

        /* elaborazione nodo */ /*4*/

        puts(p1->titolo);

        puts(p1->autore);

        puts(p1->editore);

        printf("%ld\n",p1->prezzo);

        fflush(stdin);

        while(!getchar()=='\n'); /*5*/
```

```
            /* fine elaborazione nodo */

            esamina(p1->next); /*6*/

        }/* fine if */

    }(* fine programma */
```

Nella `1` viene dichiarato il prototipo della funzione che si occuperà di esaminare un nodo della lista: tale funzione riceverà il puntatore al nodo da esaminare.

La funzione verrà chiamata, come in `2`, passandogli inizialmente l'*entry point* in modo da puntare al primo elemento.

L'esame degli elementi della lista è effettuato con una funzione ricorsiva: se il puntatore passato (vedi `3`) è un puntatore valido, elabora l'elemento e richiama la funzione passando il puntatore contenuto nel nodo esaminato (il puntatore al prossimo nodo). A cominciare da `4` si elabora l'elemento (quello raggiungibile per mezzo del puntatore `p1`); nel caso specificato in questa sede, l'elaborazione è una visualizzazione delle informazioni contenute nel nodo stesso.

La funzione `getchar` specificata in `5` legge un carattere dalla tastiera. In questa applicazione viene svuotato il buffer della tastiera (riga precedente la `5`) e si aspetta (per mezzo del ciclo presente in `5`) il carattere *newline*. In pratica si blocca il programma permettendo di leggere gli output: il programma proseguirà in seguito alla digitazione del tasto <Invio> .

Nella `6` viene chiamata ricorsivamente `esamina` passandogli come parametro il puntatore al prossimo nodo della lista.

inserimento di un nuovo nodo

La lista concatenata è un insieme ordinato di elementi: la successione logica degli elementi è quella data dai puntatori. Nel frammento di programma successivo ci si occuperà dell'inserimento di un nuovo libro nella lista in modo che i libri siano ordinati per titolo.

L'inserimento di un nuovo elemento, così come la cancellazione di un elemento esistente, sono operazioni che avvengono in modo agevole perché vengono effettuate mediante l'uso dei puntatori. Sono operazioni molto veloci sia perché per mantenere l'ordine non è necessario inserire fisicamente l'elemento nel posto che gli compete (si sottolinea che, a differenza dell'array, l'ordine logico non coincide con l'ordine fisico), sia perché si opera mediante i puntatori con le locazioni di memoria interessate alle operazioni.

Prima di passare ad esaminare le funzioni che effettuano l'inserimento di un nodo in una lista, è bene chiarire l'algoritmo utilizzato al fine di una più agevole comprensione delle funzioni stesse e dei parametri utilizzati.

Il programma si occuperà per prima cosa di trovare il posto giusto dove inserire il nodo (si ricorda che i libri devono essere registrati in ordine alfabetico per titolo).

Tenendo conto che ogni elemento della lista contiene oltre ai dati del libro registrato, il puntatore al prossimo elemento, e che è proprio questo che occorre modificare affinché punti all'elemento da inserire, occorre conoscere l'indirizzo di memoria del puntatore da modificare affinché possa avvenire tale modifica. Non si tratta infatti, in questo caso, di accedere all'elemento successivo per cui basta solo la conoscenza del valore del puntatore, bensì di modificare tale valore affinché punti ad un nuovo elemento.

Trovato il puntatore da modificare l'inserimento dell'elemento viene effettuato in maniera agevole.

```
...

/* Funzioni per la gestione della Lista */
```

```
void inserisci(lpointer *pins,lpointer pnodo); /*1*/

/* Altre funzioni del programma */

void nuovolib(); /*1*/

void aggiungi(lpointer pn); /*1*/

lpointer* cerca(lpointer *inizio,char tit); /*1*/

...

/*

Inserimento di un libro nella lista

i libri sono inseriti alfabeticamente per titolo

*/

void nuovolib()

{

    lpointer nuovo;

    nuovo=(lpointer) malloc(sizeof(libro)); /*2*/

    if(nuovo==NULL)

    {

        printf("\nMemoria esaurita: inserimento impossibile");

    }

    else

    {

        printf("\nInserimento di un libro");

        fflush(stdin);

        printf("\nTitolo : ");

        gets(nuovo->titolo);

        printf("Autore : ");

        gets(nuovo->autore);

        printf("Editore : ");

        gets(nuovo->editore);

        printf("Prezzo : ");

        scanf("%ld",&nuovo->prezzo);

        aggiungi(nuovo); /*3*/
```

```
    }/* fine else */
}/* fine funzione */



void aggiungi(lpointer pn)
{
    lpointer *ppos; /*4*/
    ppos=cerca(&entryp,pn->titolo); /*5*/
    inserisci(ppos,pn); /*6*/
}



lpointer* cerca(lpointer *inizio,char *tit)
{
    if(*inizio==NULL) /*7*/
    return inizio;
    if(strcmp((*inizio)->titolo,tit)>0) /*8*/
        return inizio;
    cerca(&((*inizio)->next),tit); /*9*/
}



/* Funzioni per la gestione di una lista */
void inserisci(lpointer *pins,lpointer pnodo)
{ /*10*/
    pnodo->next=*pins; /*11*/
    *pins=pnodo; /*12*/
}
```

Nelle 1, al solito, sono riportati i prototipi delle funzioni utilizzate: la `inserisci` si occupa dell'inserimento effettivo mentre le altre funzioni si occupano di predisporre le condizioni affinché tale inserimento possa aver luogo.

In 2 viene allocato, tramite la funzione malloc, lo spazio necessario per contenere un elemento di tipo libro. La funzione restituisce un puntatore alla memoria allocata, è necessario quindi verificare se tale puntatore punta effettivamente ad una zona di memoria valida (il puntatore deve cioè contenere un valore diverso da NULL). Se tale condizione è verificata vengono acquisiste le informazioni da conservare e si passa, nella 3, il valore contenuto nel puntatore alla funzione aggiungi che si occuperà dell'inserimento.

La funzione aggiungi dichiara nella 4 la variabile ppos come puntatore a lpointer, praticamente un puntatore a puntatore. È in questi casi che la typedef fornisce i suoi maggiori vantaggi: senza il suo uso si sarebbe dovuto scrivere struct rec **ppos; con un notevole sforzo mentale per cercare di fissare le idee sul senso di puntatori e puntatori di puntatori. Con la definizione di tipo la 4 si può leggere: si dichiara ppos come un puntatore al tipo lpointer. Come osservato prima si tratta di cercare il puntatore il cui valore dovrà essere modificato, è necessario passare alle funzioni un puntatore alla variabile il cui contenuto dovrà essere modificato.

La funzione cerca si occupa di ritornare, così come evidenziato in 1, un puntatore al dato da modificare (che è un puntatore). A tale funzione viene passato, come specificato in 5, l'entry point e il titolo da confrontare per trovare la giusta posizione di inserimento. Alla funzione inserisci, chiamata successivamente in 6, si passa il puntatore al dato da modificare ritornato da cerca e il puntatore all'elemento da inserire.

La funzione cerca verifica, in 8, se il titolo del libro puntato è alfabeticamente successivo al titolo del libro che si sta inserendo, o, in 7, se la lista è terminata. Laddove una di queste condizioni è verificata la funzione ritorna l'indirizzo del puntatore poiché è questo il dato da modificare e quindi occorre conoscere il suo indirizzo. Se nessuna delle condizioni descritte prima è verificata, la funzione chiama ricorsivamente sé stessa (9) passando ad esaminare il prossimo elemento della lista.

La funzione inserisci si occupa dell'inserimento effettivo dell'elemento nella struttura concatenata. La funzione riceve, come evidenziato in 10, il puntatore all'indirizzo da modificare e il puntatore all'elemento da inserire. Il puntatore al punto di inserimento dovrà contenere l'indirizzo del nuovo elemento (12), ma prima il vecchio contenuto del punto di inserimento dovrà essere assegnato al puntatore del nuovo elemento (11). È opportuno fare notare che un errore nell'assegnazione dei valori ai puntatori, per esempio scambiare di posto la 11 e la 12, comporta la perdita di fatto degli elementi successivi della lista: ogni elemento contiene informazioni per rintracciare il prossimo, basta modificare in maniera erronea un puntatore perché tutti gli elementi da quel punto in poi, scompaiano, anzi non sono mai esistiti (non è possibile rintracciarli in alcun modo).

eliminazione di un nodo

L'algoritmo per l'eliminazione di un nodo dalla lista è simile a quello dell'inserimento.

Si cerca l'indirizzo dell'elemento da cancellare, si modifica tale indirizzo con il contenuto del puntatore dell'elemento da cancellare (in modo che tale indirizzo non punti più all'elemento che si deve eliminare, ma al prossimo della lista) e si rende libera, per eventuali altri usi, l'area di memoria prima occupata dall'elemento cancellato.

```
...

/* Funzioni per la gestione della Lista */

void elimina(lpointer *pcanc);

/* Altre funzioni del programma */

lpointer* cerca(lpointer *inizio,char *tit);
```

```
void canclib();
...
/* Cancella un titolo dalla lista */
void canclib()ù
{
    char titcanc[50];
    lpointer *pdel;
    fflush(stdin);
    printf("\nTitolo da cancellare :");
    gets(titcanc); /*1*/
    pdel=cerca(&entryp,titcanc); /*2*/
    if(*pdel==NULL) /*3*/
        printf("\nTitolo inesistente: cancellazione impossibile");
    else
        elimina(pdel); /*4*/
}


lpointer* cerca(lpointer *inizio,char *tit)
{
    ...
    if(!strcmp((*inizio)->titolo,tit)) /*5*/
        return inizio;
    ...
}


/* Funzioni per la gestione di una lista */
void elimina(lpointer *pcanc)
{
    lpointer temp;
```

```
        temp=*pcanc; /*6*/

        *pcanc=(*pcanc)->next; /*7*/

        free(temp); /*8*/

    }
```

In 1 viene acquisito il titolo del libro da eliminare e passato, in 2, come parametro alla funzione che si occuperà di cercare il puntatore all'elemento da cancellare. Se tale puntatore contiene un valore valido (test effettuato a cominciare da 3), si passa come parametro (in 4) alla funzione `elimina` che procederà alla cancellazione dell'elemento dalla lista.

La funzione `cerca` è la stessa funzione esaminata nell'inserimento: cambia solo il confronto da effettuare (espresso in questo caso dalla 5). In questo caso si tratta di trovare la coincidenza del titolo conservato nel nodo con il titolo da cancellare.

L'eliminazione dell'elemento prevede in 6 la conservazione temporanea del puntatore al fine di recuperare l'area di memoria con la chiamata alla `free` della 8. La cancellazione dell'elemento è effettuata in 7: nella sequenza dei puntatori si salta l'elemento da non considerare aggiornando in tal modo la sequenza logica degli elementi.

L'input-output astratto

Le periferiche disponibili in un sistema di elaborazione sono, dal punto di vista hardware, anche molto diverse fra di loro anche se dal punto di vista dell'utente le funzioni che svolgono possono essere assimilabili. Si pensi, per esempio, alle differenze sostanziali fra una stampante ad aghi e ad una laser: dal punto di vista dell'utente si tratta in tutte e due i casi di una stampante (una periferica che produce un output su supporto cartaceo), dal punto di vista hardware si tratta di due cose distinte almeno quanto lo potrebbero essere, per esempio, una stampante ad aghi e una tastiera.

Il sistema operativo fornisce un'interfaccia ad alto livello verso l'hardware: le periferiche sono mappate in memoria, è utilizzata cioè in pratica una parte della memoria centrale (il *buffer*) come deposito temporaneo dei dati da e verso le periferiche. In questo modo, per esempio, le operazioni di input possono essere effettuate sempre allo stesso modo a prescindere dalla periferica: sarà il sistema che si occuperà della gestione della specificità dell'hardware. Il sistema di I/O fornisce il concetto astratto di **canale** (lo *stream*). Con tale termine si intende un dispositivo logico indipendente dalla periferica fisica: chi scrive il programma si dovrà occupare dei dati che transitano per il canale prescindendo dalle specifiche del dispositivo fisico che sta usando (un lettore di dischi magnetici, una stampante). Il termine **file** si riferisce invece ad una astrazione che è applicata a qualsiasi tipo di dispositivo fisico. In poche parole, si potrebbe affermare che il file rappresenta il modo attraverso il quale l'utilizzatore vede sistemati i dati sul dispositivo di I/O, e che il canale è il modo con cui i dati sono accessibili per l'utilizzazione.

Un canale è associato ad un file per mezzo di una `open` (apertura del file), a questo punto i dati presenti nel file sono accessibili per l'utilizzo. L'associazione fra canale e file è eliminata per mezzo di una `close` che interrompe le comunicazioni. Se il canale è stato aperto per operazioni di output, la chiusura consente di scrivere sul dispositivo fisico i dati ancora presenti nel buffer (lo *svuotamento del canale*). In questo modo si evita che una parte dei dati possa non essere registrata.

L'associazione di canali alla tastiera per l'input e al video per l'output è curata in automatico dal sistema operativo per consentire il dialogo con il sistema. La tastiera e il video sono cioè le *periferiche di default*: il sistema è già connesso con esse. Per quanto riguarda invece le comunicazioni con altre periferiche è necessario esplicitare l'associazione di canali per tali comunicazioni.

inizio

I dati su memorie di massa: files sequenziali

Le strutture dati trattate fino a questo punto, prevedono la registrazione degli elementi su locazioni di memoria centrale. La memoria centrale, in ragione delle sue caratteristiche, si presta a due tipi di utilizzo:

- Per la sua **limitatezza** è opportuno utilizzarla per la conservazione dei dati strettamente indispensabili per l'elaborazione in corso.
- Per la sua **volatilità** consente di conservare i dati limitatamente al solo tempo di esecuzione di un programma.

Se si vogliono conservare grandi quantità di dati e utilizzarli in tempi diversi, è necessario affidarsi a supporti permanenti che non hanno le limitazioni della memoria centrale: sono quelli che comunemente sono chiamate memorie di massa.

Nell'esempio successivo si gestiranno una serie di libri, conservando in un file su memoria di massa i dati sugli stessi. I dati sui libri saranno registrati uno di seguito all'altro: quando si tratterà di rileggerli, l'ordine di reperimento delle informazioni sarà lo stesso di quello che si è utilizzato per la scrittura. I dati saranno quindi elaborati in maniera sequenziale.

```
/* Gestione File sequenziale */

#include <stdio.h>

/* Definizione record */

typedef struct

{

    char titolo[50];

    char autore[20];

    char editore[20];

    long int prezzo;

}libro;


/* Funzioni per la gestione del file */

void inserisci();

void esamina();


main()

{

    int scelta;

    for(;;)

    {

        /* Menu operazioni disponibili */

        printf("\nGestione di un file contenente libri\n");

        printf("\n1) Inserimento di un libro nel file");

        printf("\n2) Esame dei libri contenuti nel file");

        printf("\n0) Fine elebaorazione\n");

        printf("\nScelta operazione (1..2,0) ");

        scanf("%d",&scelta);

        if(!scelta)

            break;


        /* Richiama funzione scelta */
```

```
        switch(scelta)
        {
        case 1: inserisci();
        break;

        case 2: esamina();
        }/* fine switch */
    }/* fine ciclo for */
    return 0;
}/* fine programma */



/* Inserimento di un libro nel file */
void inserisci()
{
    libro buflib; /*1*/
    FILE *fp; /*2*/
    fp=fopen("LibriSeq.dat","a"); /*3*/
    if(fp==NULL) /*4*/
        return;

    printf("\nInserimento di un libro");
    fflush(stdin);
    printf("\nTitolo : ");
    gets(buflib.titolo);
    printf("Autore : ");
    gets(buflib.autore);
    printf("Editore : ");
    gets(buflib.editore);
    printf("Prezzo : ");
```

```
    scanf("%ld",&buflib.prezzo);

    /* conservazione nel file */
    fputs(buflib.titolo,fp); /*5*/
    fprintf(fp,"\n");
    fputs(buflib.autore,fp);
    fprintf(fp,"\n");
    fputs(buflib.editore,fp);
    fprintf(fp,"\n");
    fprintf(fp,"%ld\n",buflib.prezzo);
    fclose(fp); /*6*/
}/* fine funzione */



/* Scansione sequenziale del file dei libri */
void esamina()
{
    libro buflib;
    FILE *fp;
    fp=fopen("LibriSeq.dat","r"); /*7*/
    if(fp==NULL)
        return;

    for(;;)
    {
        if (fgets(buflib.titolo,50,fp)==NULL) /*8*/
            break;
        fgets(buflib.autore,20,fp); /*9*/
        fgets(buflib.editore,20,fp);
        fscanf(fp,"%ld",&buflib.prezzo);
```

```
            fgetc(fp); /*10*/

            puts(buflib.titolo);

            puts(buflib.autore);

            puts(buflib.editore);

            printf("%ld",buflib.prezzo);

            fflush(stdin);

            while(!getchar()=='\n');

        }/* fine for */

        fclose(fp);

    }/* fine funzione */
```

Nella 1 viene definito il buffer che conterrà il libro da registrare nel file.

Nella riga 2 viene dichiarato un puntatore (*file pointer*) alla struttura FILE. Tale struttura, definita in stdio.h, in accordo con quanto espresso prima, rappresenterà il canale associato al file nel quale saranno conservati i dati sui libri.

Nella 3 viene effettuata una chiamata alla funzione fopen. Tale chiamata ritorna un puntatore al tipo FILE. I parametri da passare alla funzione richiedono di specificare una stringa contenente il nome con il quale il file è registrato nella memoria di massa e una stringa specificante la modalità di apertura del file. In questo caso il file è aperto in modalità *append*: i record saranno registrati di seguito a quelli già presenti nel file. Se il file non esiste verrà creato.

Il puntatore ritornato dalla fopen può essere NULL se il sistema non ha potuto generare, per un motivo qualsiasi, il file richiesto (per esempio se il dischetto è protetto da scrittura o se non c'è più spazio). Tale eventualità è testata nella 4.

A cominciare da 5 per scrivere i dati sul file, così come messo in evidenza prima, vengono utilizzate le funzioni fputs e fprintf formalmente uguali alle puts e printf più volte utilizzate. La funzione fputs, a differenza della puts, richiede di specificare oltre alla stringa da scrivere, la specifica del canale attraverso il quale effettuare l'operazione. Una osservazione simile vale per la fprintf, solo che stavolta il canale va specificato come primo parametro. Ogni registrazione di un singolo dato è seguita dalla registrazione di un carattere *newline* al fine di un più semplice reperimento dei singoli dati in fase di lettura. Per maggiori chiarimenti si legga quanto osservato nella funzione di lettura dal file.

La funzione fclose della 6 si occupa di interrompere le comunicazioni con il file.

Per poter accedere ai dati contenuti nel file, è necessario aprirlo in modalità *read*. Questo è ciò di cui si occupa l'istruzione contenuta nella riga 7.

La funzione esamina legge tutti i record registrati nel file e li visualizza sul video. A tal fine usa, come specificato in 8 e 9 le funzioni fgets e fscanf. La funzione fgets necessita, oltre che della specifica della stringa dove depositare la lettura, della quantità massima di caratteri da leggere e del canale attraverso il quale effettuare la lettura. È opportuno notare che la funzione legge dal file fino al numero di caratteri specificato o al *newline* se questo viene incontrato prima. Questo è il motivo dell'inserimento di tale carattere come delimitatore dei singoli dati: non si può infatti conoscere l'esatta lunghezza del dato contenuto nel campo e così si utilizza il *newline* come delimitatore. La funzione fgetc della 10 legge l'ultimo *newline* inserito dopo l'ultima fprintf in sede di scrittura.

L’elaborazione sequenziale di un file prevede la lettura di tutte le registrazioni contenute in esso così come sono state registrate. Per verificare l’avvenuta lettura di tutte le registrazioni contenute nel file, si è inserito il controllo specificato in `8`. Se i libri registrati sono stati tutti letti, un ulteriore tentativo di lettura di una stringa ritorna un puntatore `NULL`.

inizio

Files ad accesso casuale

Se il supporto sul quale sono registrati i dati è gestito da un dispositivo che lo consente, è possibile accedere ad una registrazione qualsiasi contenuta nel file, specificando la sua posizione relativa all’interno del file stesso. La logica di gestione di un file ad accesso casuale somiglia a quella conosciuta della gestione di una tabella. Anche in quel caso i records erano accessibili specificando la loro posizione relativa. Spingendo la similitudine con gli argomenti trattati in precedenza si può dire che i records in un file sequenziale sono elaborati come gli elementi di una coda, in un file ad accesso casuale come una tabella.

```
/*
Gestione file di libri con elaborazione sequenziale
ed accesso diretto
*/
#include <stdio.h>
/* Definizione del record */
typedef struct
{
    char titolo[50];
    char autore[20];
    char editore[20];
    long int prezzo;
}libro;
...



/* Inserimento di un libro nel file */
void inserisci()
{ /*1*/
    FILE *fp;
    libro buflib;
    fp=fopen("DatiLib.dat","a"); /*2*/
```

```
    if(fp==NULL)
        return;
    printf("\nInserimento di un libro");
    fflush(stdin);
    printf("\nTitolo : ");
    gets(buflib.titolo);
    printf("Autore : ");
    gets(buflib.autore);
    printf("Editore : ");
    gets(buflib.editore);
    printf("Prezzo :");
    scanf("%ld",&buflib.prezzo);
    fwrite(&buflib,sizeof(libro),1,fp); /*3*/
    fclose(fp);
}/* fine funzione */



/* Estrazione di un libro dal file */
void estrai()
{ /*4*/
    FILE *fp;
    libro buflib;
    int quale;
    long dove;
    fp=fopen("DatiLib.dat","r"); /*5*/
    if(fp==NULL)
        return;
    printf("\nEstrazione di un libro dal file");
    printf("\nQuale libro :");
    scanf("%d",&quale); /*6*/
```

```
    dove=(long) sizeof(libro)*(quale-1); /*7*/
    fseek(fp,dove,SEEK_SET); /*8*/
    if(!fread(&buflib,sizeof(libro),1,fp))
    { /*9*/
        printf("\nLibro inesistente");
        return;
    }/* fine if */
    puts(buflib.titolo);
    puts(buflib.autore);
    puts(buflib.editore);
    printf("%ld\n",buflib.prezzo);
    fflush(stdin);
    while(!getchar()=='\n');
    fclose(fp);
}/* fine funzione */



/* Scansione sequenziale del file */
void esamina()
{ /*10*/
    FILE *fp;
    libro buflib;
    fp=fopen("DatiLib.dat","r");
    if(fp==NULL)
        return;
    printf("\nLibri conservati nel file\n");
    fread(&buflib,sizeof(libro),1,fp); /*11*/
    while(!feof(fp))
    { /*12*/
        puts(buflib.titolo);
```

```
            puts(buflib.autore);

            puts(buflib.editore);

            printf("%ld\n",buflib.prezzo);

            fflush(stdin);

            while(!getchar()=='\n');

            fread(&buflib,sizeof(libro),1,fp); /*11*/

        }/* fine while */

        fclose(fp);

    }/* fine funzione */
```

La funzione `inserisci` definita in `1` si occupa, allo stesso modo dell'equivalente nell'esempio del file sequenziale, della conservazione di un libro nel file. Formalmente è molto simile all'altra vista in precedenza. Anche in questo caso c'è un buffer per la conservazione temporanea del record da registrare e un puntatore alla struttura `FILE`. Anche l'apertura del file viene effettuata, come si legge in `2`, per mezzo di una chiamata alla funzione `fopen`.

Per quanto riguarda la srittura dei dati sul file, in questo caso, viene utilizzata nella `3` la funzione `fwrite` che si occupa di scrivere un blocco di byte. Tale funzione richiede come parametri un puntatore all'area da cui prelevare i dati da scrivere (`&buflib`), il numero di byte che devono essere scritti (`sizeof(libro)`), quanti blocchi di quella dimensione scrivere (`1`), il canale da utilizzare (`fp`).

La funzione `estrai` della `4`, dopo aver aperto in `5` il file similmente alla equivalente nel file sequenziale, si occupa di rintracciare una specifica registrazione all'interno del file, una volta conosciuta la dimensione comune dei records conservati nel file e la posizione del record interessato. Il metodo utilizzato, per il rintracciamento del record interessato, è lo stesso di quello utilizzato per il rintracciamento di un elemento in una struttura sequenziale (indirizzo base, scostamento da effettuare): nella `6` viene richiesto di specificare la posizione del record da leggere, nella `7` tale posizione relativa viene trasformata in indirizzo relativo. Si noti che l'espressione qui utilizzata è l'equivalente della seconda parte di $IND(x_i)=IND_b+(i-1)l$, infatti qui la dimensione comune degli elementi è rappresentata da `sizeof(libro)`. L'espressione per esteso è scritta nella `8`, dove la funzione `fseek` ci posiziona sul record interessato. La funzione richiede infatti come parametri oltre che il puntatore al file (`fp`) l'indirizzo base da cui partire (`SEEK_SET` è il simbolo utilizzato per indicare l'inizio del file ed è definito in `stdio.h`), lo scostamento relativo calcolato in precedenza. Si noti che è semplicemente un modo diverso di scrivere l'espressione ricordata prima.

Effettuato il posizionamento, la funzione `fread` della `9` si occupa di depositare nel buffer predisposto (l'area di memoria associata a `&buflib`) il record letto. La funzione richiede gli stessi parametri della `fwrite` utilizzata in `3`. La funzione ritorna un puntatore all'area utilizzata per conservare il record letto (lo stesso valore di `&buflib`). Se l'operazione di lettura non ha avuto esito positivo, per esempio perché non esiste un record registrato all'indirizzo specificato, il puntatore assume valore `NULL`: tale condizione è testata appunto nella `9`.

I record conservati in un file ad accesso casuale possono essere elaborati anche sequenzialmente ed è di questo tipo di elaborazione che si occupa la funzione `esamina` della `10`. Anche in questo caso, dopo aver effettuato le solite operazioni, si procede alla lettura di tutti i record contenuti nel file nello stesso ordine in cui sono registrati: a tale fine è utilizzata la funzione `fread` delle `11`. La prima volta del suo utilizzo, poiché non è stata utilizzata la `fseek`, viene letto il primo record registrato. Ogni nuova chiamata alla `fread` ritorna in `buflib` il record successivo e in questo modo possono essere letti tutte le registrazioni contenute nel file. Non conoscendo la quantità di record registrati nel file, è necessario utilizzare la funzione `feof`. Tale funzione

ritorna il valore di verità (`1`) se nell'ultima operazione di lettura, dal canale specificato come parametro (`fp` nell'esempio), si è raggiunti la fine del file.