

FONDAMENTI DI INFORMATICA

**Corsi di Laurea in Ingegneria Elettrica,
Ingegneria Gestionale, Ingegneria Meccanica**

Università degli Studi di Firenze

A. A. 2001-2002

APPUNTI DELLE LEZIONI

Terza parte

Prof. Alessandro Fantechi

1. L'architettura di un sistema di elaborazione:

In questi appunti illustriamo gli aspetti più importanti di un sistema di elaborazione, descrivendo per grandi linee l'architettura hardware nota come "architettura di Von Neumann", architettura alla base dei sistemi di elaborazione odierni, ed il suo funzionamento elementare.

In tale architettura (fig.1) si possono individuare le seguenti unità:

- *memoria centrale* (spesso indicata come RAM) , che ha lo scopo di memorizzare programmi, dati di ingresso, risultati intermedi e finali;
- *unità centrale di elaborazione* (CPU - Central Processing Unit) o processore
- *unità di ingresso e uscita*, (I/O - input/output) che svolge le funzioni di interfaccia tra l'elaboratore e l'esterno.

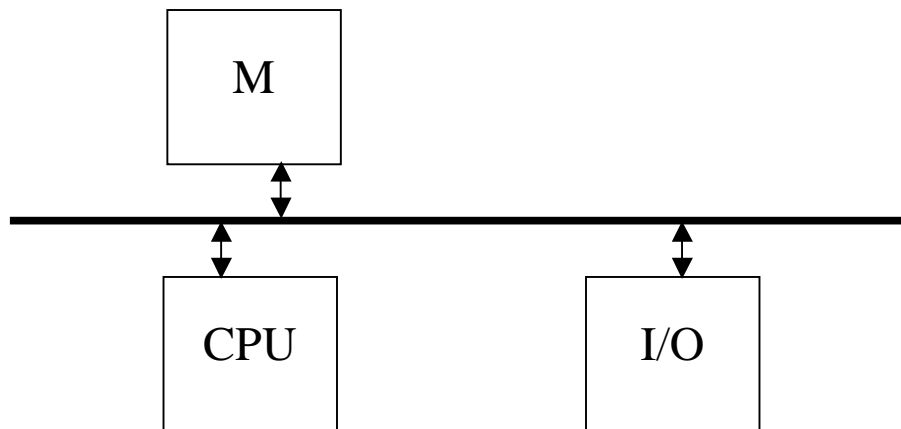


Fig. 1 - Schema funzionale dell'elaboratore

1.1 La memoria centrale

Una sequenza di 8 bit prende il nome di *byte*; un byte può quindi rappresentare 2^8 diversi elementi. La memoria di un elaboratore elettronico è costituita da un insieme di celle o locazioni costituite da n bit, dove n è un valore fisso per ogni elaboratore. Esistono elaboratori con celle di 4, 8, 16, 32, 36, 64... bit. Il contenuto di una cella di memoria prende il nome di *parola*.

Ad ogni cella di memoria è associato un *indirizzo* che è il numero d'ordine della cella nell'intera memoria. Ad ogni cella corrisponde pertanto la coppia <indirizzo, valore>.

Accedere ad un dato in memoria significa selezionare mediante l'indirizzo la cella in cui esso è memorizzato e prelevarne il valore. Analogamente, la memorizzazione di un dato richiede di stabilire l'indirizzo della cella in cui si intende introdurlo.

Se una cella della memoria è formata da un numero di bit multiplo di otto, come accade normalmente nei moderni computer, significa che in essa sono contenuti più byte e spesso è possibile accedere direttamente sia alla parola intera che a ciascun byte in essa contenuto: in questo caso si dice che la memoria è *indirizzata al byte*.

Una caratteristica importante della memoria è il tempo di accesso, cioè il tempo che intercorre tra l'istante in cui si richiede un'informazione e l'istante in cui tale informazione è disponibile. La memoria centrale viene anche detta ad accesso diretto (*Random Access Memory: RAM*) o uniforme, poiché il tempo di accesso è costante, indipendentemente dall'indirizzo della cella a cui si vuole accedere. Il tempo di accesso è, nell'attuale tecnologia, dell'ordine di alcune centinaia di nanosecondi (10^{-7} sec).

Un altro parametro che caratterizza la memoria centrale, è il numero complessivo di celle di cui essa è formata, che può andare dalle decine di migliaia alle centinaia di milioni. Nei computer

odierni la memoria si misura in MB (*megabyte*), cioè milioni di byte. In realtà, il kilobyte (KB) equivale a 1024 byte (cioè 2^{10}) e il megabyte equivale a $1024 * 1024 = 1048576 = 2^{20}$ byte. Il Gigabyte (GB), e i Terabyte (TB) unità di misura utilizzate per dischi o per sistemi di dischi, valgono rispettivamente $1024^3 = 2^{30}$ byte e $1024^4 = 2^{40}$ byte

In una memoria di 16 MB indirizzata al byte, vi sono quindi 2^{24} locazioni di un byte, indirizzabili quindi con numeri da 0 a $2^{24}-1$. Ognuno di questi indirizzi è quindi rappresentabile con 24 bit.

Un tipo particolare di memoria centrale è costituito dalle memorie a sola lettura (*Read Only Memory: ROM*) nelle quali non è possibile memorizzare nuovi valori nelle celle. Le memorie ROM sono perciò utilizzate specificamente per contenere programmi e dati che non vengono mai modificati durante l'elaborazione.

Si noti che le memorie ROM sono permanenti, mentre per quelle RAM il contenuto delle celle viene alterato ad ogni interruzione della corrente di alimentazione dell'elaboratore, cioè sono *volatili*.

1.2 Unità centrale di elaborazione

L'unità centrale di elaborazione (CPU) o processore è l'elemento effettivamente capace di eseguire gli algoritmi che vengono scritti, sotto forma di programmi, in memoria.

I moderni PC usano processori di una stessa famiglia, di provenienza Intel, a partire dai primi 8086, per seguire con gli 80286, 80386, 80486 e i Pentium. I computer Apple (Macintosh, Imac, etc..) usano invece un processore di una famiglia di provenienza Motorola, a partire dal 68000, poi 68020, 68030, 68040, PowerPC, G3, G4, etc....

I processori di una stessa famiglia condividono la stessa architettura, cioè la struttura e i principali componenti interni, e il set di istruzioni, e si differenziano tra loro per potenza di calcolo (cioè velocità di elaborazione), tecnologia usata, utilizzo di tecniche più o meno sofisticate per aumentare le prestazioni.

La CPU è composta da tre parti principali: una *unità aritmetico-logica* (ALU), un insieme di *registri* e una *unità di controllo*. La ALU esegue operazioni aritmetiche (per esempio addizioni e moltiplicazioni), operazioni logiche e confronti (per esempio $<$) sui dati della memoria principale o dei registri del calcolatore. La ALU contiene quindi un sommatore e altre reti combinatorie necessarie per eseguire operazioni.

I *registri* sono degli spazi di memorizzazione che consentono un accesso molto veloce ai dati che vi sono contenuti (tempo di accesso dell'ordine dei nanosecondi - 10^{-7} sec): in un processore moderno vi sono un numero di registri dell'ordine della decina: L'insieme dei registri è spesso indicato con il nome di "registri generali" per distinguerli da altri registri presenti all'interno dell'unità centrale, con funzioni ausiliarie. Ciascun registro è di solito composto da un numero di bit pari a quello di una parola di memoria, in modo da poter trasferire un dato dalla memoria a un registro e viceversa.

L'*unità di controllo* governa il funzionamento complessivo del sistema di elaborazione attraverso il controllo dell'esecuzione delle singole istruzioni di un programma. Come vedremo più avanti, le istruzioni del programma vengono trasferite, una alla volta, dalla memoria all'unità di controllo, che ne garantisce l'esecuzione inviando comandi opportuni alle altre unità del sistema;

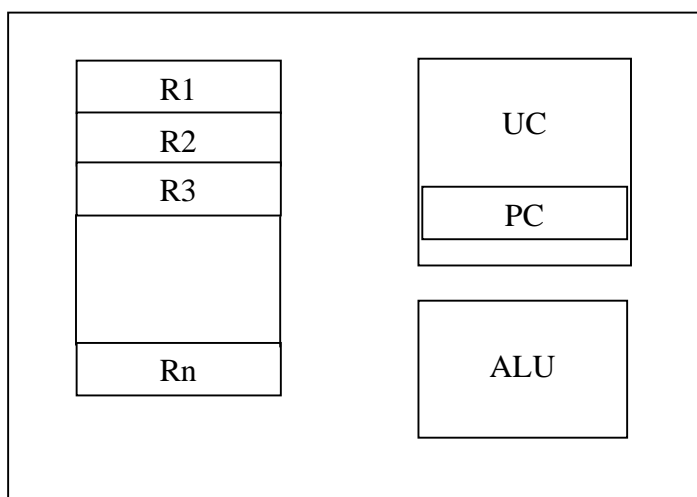


Fig. 2 - Struttura interna dell'unità centrale

1.3 Le unità di ingresso e uscita

Le unità di ingresso e uscita svolgono le funzioni di interfaccia tra l'elaboratore e l'esterno. In effetti, l'esecuzione degli algoritmi avviene nel complesso Processore/Memoria, e le unità di I/O vengono interessate solo quando si debba ricevere dati dall'esterno o spedire dati all'esterno.

Vi possono essere vari tipi di dispositivi di I/O: tastiera, mouse, monitor, stampanti, modem, scanner, etc...; si considerano però come tali anche quei dispositivi di memorizzazione che formano la cosiddetta *memoria secondaria*, quali dischi rigidi (hard disk), floppy disk, CD, etc...

Nonostante quindi questi dispositivi ospitino la maggior parte delle informazioni contenute in un sistema, non sono considerati parte del cuore del computer, e vengono acceduti (cioè vi si leggono e scrivono dati) esattamente come si accede ad un modem o a una stampante.

Di solito i dispositivi di I/O sono fisicamente esterni al processore (anche i dischi rigidi e CD che sono montati nel "case" del computer sono esterni al complesso Processore/Memoria, che si trova su una scheda detta "scheda madre" (motherboard). Su questa scheda, collegata al bus, vi sono una o più interfacce verso i dispositivi di I/O, che vi sono collegati attraverso collegamenti di vario tipo (bus IDE, linea USB, linea SCSI, etc...). E' quindi corretto schematizzare l'I/O come un singolo blocco di contenuto variabile, attaccato al bus, come in Fig. 1.

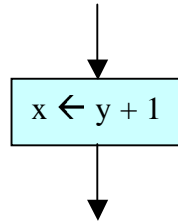
La sommara descrizione dell'architettura di un calcolatore qui fatta non prende in considerazione una miriade di varianti e dispositivi aggiuntivi che possono far parte dell'unità centrale, o dei dispositivi di ingresso uscita., che da un punto di vista concettuale non sono necessari, ma che spesso producono significative differenze prestazioni e di prezzo.

1.4 Esecuzione degli algoritmi

Abbiamo studiato nelle parti precedenti del corso come sia possibile definire algoritmi di risoluzione di problemi, e come sia possibile rappresentare all'interno di una macchina costituita di componenti elettronici informazioni di qualsiasi tipo. Abbiamo anche visto, senza addentrarci nei dettagli, come tali macchine possano compiere operazioni su tali informazioni.

A questo punto manca però un passo essenziale: come poter eseguire un algoritmo? Per dare una risposta, pur non certo dettagliata, a questa domanda possiamo riconsiderare alcuni degli algoritmi visti, soffermandoci in particolare dapprima sulla esecuzione di una singola istruzione.

Consideriamo ad esempio il seguente blocco di un diagramma di flusso:



Questo blocco può essere fatto corrispondere all'istruzione in linguaggio C:

$$x = y + 1;$$

Per eseguire questa istruzione occorrerà innanzitutto avere a disposizione delle zone della memoria che utilizzeremo per rappresentare le variabili x e y . Supponiamo che si siano dedicate a queste variabili rispettivamente le locazioni di indirizzo 588 e 256 (si dice che le variabili x e y sono *allocate* in memoria all'indirizzo rispettivamente 588 e 256).

Per eseguire l'istruzione, il processore dovrà in realtà eseguire l'istruzione:

$$M[588] \leftarrow M[256] + 1$$

attraverso i seguenti passi:

- leggere da memoria all'indirizzo 256
- sommare 1 al contenuto
- scrivere in memoria il risultato all'indirizzo 588

Quindi, se supponiamo che la locazione di memoria all'indirizzo 256 contenga il valore 3014, potremmo osservare i seguenti valori viaggiare sul bus tra CPU e memoria:

CPU – 256 → M (indirizzo della richiesta di lettura da memoria)

M – 3014 → CPU (valore letto dalla memoria)

CPU – 588 → M (indirizzo della richiesta di scrittura in memoria)

CPU – 3015 → M (valore scritto in memoria)

Il numero dei bit di cui è composto il bus limita i valori che possono transitare tra memoria e CPU. Si noti però che alcuni dei valori sopra indicati sono degli indirizzi. Se la memoria fosse, per esempio, di 16MB, cioè di 2^{24} byte, gli indirizzi sarebbero valori da 0 a $2^{24}-1$; pertanto, dovrebbero essere rappresentati su 24 bit, e questa dovrebbe essere la minima “larghezza” del bus.

In quanto abbiamo detto finora, possiamo notare che una variabile del diagramma di flusso (o del programma C) è un contenitore di dato che ha associato un nome (o identificatore) che la identifica in modo univoco. La realizzazione di questa variabile è necessariamente ottenuta come un elemento di memoria, formato da una o più locazioni a seconda della dimensione della variabile in termini di bit, e pertanto in funzione del suo tipo. Quindi, ad ogni variabile viene associato anche l'indirizzo della locazione di memoria che la realizza.

Poniamo attenzione al fatto che finora abbiamo considerato che un processore possa eseguire un comando scritto nel linguaggio dei diagrammi di flusso oppure scritto in linguaggio C. Purtroppo, non è facile progettare un processore che sia in effetti capace di eseguire direttamente programmi scritti nel linguaggio C.

E' notevolmente più conveniente progettare e realizzare un processore che è in grado di eseguire programmi scritti in un linguaggio apposito, e costruire dei programmi di traduzione, detti compilatori, dal linguaggio C (o da altri linguaggi di programmazione ad alto livello) a tale linguaggio.

1.5 Il linguaggio macchina

Ogni elaboratore è in grado di interpretare ed eseguire un insieme finito di istruzioni elementari, ciascuna delle quali si riferisce ad un'operazione di base che una particolare unità può compiere. Questo insieme di istruzioni costituisce il *linguaggio macchina* dell'elaboratore considerato.

Un programma eseguibile da un dato elaboratore è quindi un insieme ordinato di queste istruzioni, che vengono eseguite, una alla volta, sequenzialmente, secondo l'ordine specificato dal programma stesso. Alcune di queste istruzioni (ad esempio le istruzioni di *salto*) dovranno però poter modificare questo *flusso di controllo* sequenziale, in modo da ritornare ad istruzioni già eseguite, quando si voglia eseguire un ciclo, o in modo da saltare a istruzioni successive, per realizzare un flusso di controllo condizionato dal risultato di una decisione.

Le istruzioni che formano un programma in linguaggio macchina dovranno essere rappresentate in memoria attraverso sequenze di bit che saranno interpretate da una unità interna al processore, detta Unità di Controllo. Il processore, per seguire la sequenza di istruzioni da eseguire, dovrà contenere internamente un apposito registro, detto Program Counter (PC), che memorizza l'indirizzo della istruzione che si sta eseguendo. Alla fine dell'esecuzione di ogni istruzione, per passare a quella successiva il Program Counter verrà incrementato per andare a leggere dalla memoria la prossima istruzione.

Se consideriamo un'istruzione semplice, ad esempio una operazione di assegnamento che operi l'assegnamento:

$M[256] \leftarrow R1 + 1$ (dove per R1 si intende il contenuto del registro 1),

l'istruzione dovrà essere rappresentata come una sequenza di bit, che dovrà contenere sia una rappresentazione dell'indirizzo 256, sia una rappresentazione del numero del registro. Quindi la sequenza di bit che rappresenta l'istruzione dovrà almeno essere più lunga della rappresentazione di un indirizzo, tanto da poter essere necessario memorizzare un'istruzione in più celle di memoria adiacenti. Le regole che stabiliscono la rappresentazione di una istruzione in termini di bit definiscono quello che viene detto il *formato* delle istruzioni, in cui i bit si raggruppano, e ogni gruppo rappresenta uno dei parametri dell'istruzione, come gli indirizzi degli operandi, o il codice operativo, ovvero una configurazione di bit che decide qual è l'istruzione che si vuole eseguire (somma, sottrazione, trasferimento, salto...)

L'unità di controllo esegue le istruzioni del programma secondo quello che viene detto ciclo *accesso-decodifica-esecuzione*. Si tratta di una sequenza di passi nella quale:

- (1) si accede all'istruzione successiva (identificata tramite un indirizzo contenuto nel registro Program Counter) portandola dalla memoria in un opportuno registro interno alla UC, detto registro istruzione;
- (2) si decodifica l'istruzione;
- (3) si individuano i dati usati dall'istruzione;
- (4) si esegue l'istruzione, con l'eventuale ausilio della ALU;
- (5) si aggiorna il Program Counter (incrementandolo, o scrivendoci un nuovo valore nel caso di istruzione di salto).

Questi passi vengono ciclicamente ripetuti dal processore, in modo cadenzato da un orologio (*clock*), la cui frequenza determina la velocità del processore. Oggi è comune trovare calcolatori con una frequenza di lavoro di circa 1 GHz, per i quali il *ciclo di clock* è di 1 nanosecondo.

Un programma in linguaggio macchina è quello che si ottiene, come file .exe in Windows, quando si compila un programma scritto in linguaggio C.

Far partire l'esecuzione di un programma, cioè quello che succede quando si fa un doppio clic sull'icona di una applicazione dall'ambiente Windows, significa effettuare i seguenti passi:

- 1) copiare dal disco alla memoria il programma eseguibile
- 2) eseguire un'istruzione di salto all'inizio del programma appena caricato in memoria

Questi passi vengono compiuti da un apposito programma, che fa parte del sistema operativo, che

viene detto *loader* (caricatore).

1.6 Gerarchie di memoria

Abbiamo visto come la memoria di un sistema di elaborazione consta della memoria centrale e della memoria secondaria, o memoria di massa.

Una delle differenze principali tra questi due tipi di memoria è che le dimensioni della memoria centrale sono limitate, ma la velocità di accesso è molto elevata rispetto a quella delle memorie secondarie. Queste ultime, di dimensioni teoricamente illimitate, hanno però tempi di accesso notevolmente superiori. In particolare, nel caso di dischi, il tempo di accesso non è più indipendente dalla locazione del dato che si vuole leggere. Un disco è una superficie piana coperta di materiale ferromagnetico; sul disco le informazioni vengono organizzate all'interno di tracce, sezioni concentriche della superficie del disco. Le tracce vengono lette da una testina che può scorrere lungo un raggio del disco, in modo da poter leggere tutte le tracce. Il disco è continuamente in rotazione, per cui nel tempo di un giro completo la testina è in grado di leggere tutta la traccia.

In conclusione, il tempo di accesso a un dato su un disco dipende dalla sua posizione.

Per la loro lentezza, i dischi non possono venir usati come memoria di lavoro per il processore, che ha velocità molto superiori. Per cui di tanto in tanto vengono portati blocchi di dati (interessanti per la computazione che si deve eseguire) dal disco alla memoria centrale e viceversa; in questo modo le operazioni vengono svolte alla velocità propria del processore. Tale velocità però è superiore anche alla velocità di accesso alla RAM: il processore può avere un ciclo di clock di pochi nanosecondi, mentre l'accesso alla memoria richiede qualche decimo di microsecondo. Per questo i processori sono dotati di registri, che hanno tempi di accesso compatibili con la velocità del processore. I registri sono usati, con tecniche varie, per minimizzare l'uso della RAM in modo da aumentare la velocità di elaborazione. Anche in questo caso il principio è quello di avere una memoria piccola (e di costo per bit superiore) veloce in cui si tengono i dati più frequentemente necessari, mentre si tengono in una memoria di maggiori dimensioni, ma più lenta, i dati necessari meno di frequente, ottenendo così un buon compromesso tra velocità di elaborazione e costi.

Possiamo riassumere nella seguente tabella gli ordini di grandezza dei dati relativi a questa struttura gerarchica della memoria. Questi dati sono costantemente variati nel corso degli anni, nella direzione di un minore tempo di accesso, un minore costo a bit e quindi una maggiore dimensione.

Tipo di memoria	tempo di accesso	costo a bit	dimensione	permanenza
Registri	10^{-9} sec	10^{-2} euro	10^2 byte	volatile
RAM	10^{-7} sec	10^{-7} euro	10^8 byte	volatile
Dischi	10^{-4} sec	10^{-9} euro	10^{10} byte	permanente

Inoltre, nei moderni processori, è molto diffusa l'introduzione di un ulteriore livello di memoria, detto *cache*, di caratteristiche intermedie tra quelle della RAM e quelle dei registri, che viene utilizzato per mantenere i dati della RAM che sono usati più di frequente.

Un ulteriore livello si può anche individuare nelle tecniche di *back-up* e di archiviazione, con cui periodicamente si copia il contenuto dei dischi (o di parti di esso) su nastri, cartucce di vario tipo, o CD, con il doppio scopo di salvaguardare i dati da possibili guasti ai dischi e di scaricare i dischi da dati che si vogliono mantenere ma che non si usano correntemente. In questo caso il tempo di accesso ai dati è condizionato dall'intervento manuale di montaggio dei cosiddetti "*volumi*" (nastri, CD, etc...) nel sistema.

2. Il linguaggio C

Il linguaggio C è un linguaggio di programmazione ad alto livello. Per alto livello si intende che per scrivere programmi non è necessario conoscere le caratteristiche dell'hardware che sto usando, perché il programma è indipendente dai dettagli della struttura interna del processore. Altri linguaggi ad alto livello molto diffusi sono FORTRAN, Basic; Pascal, Ada, C++, Java.

Alcune caratteristiche tipiche del linguaggio C sono elencate qui sotto:

- Implementazione di una vasta gamma di dati
- Ristretto numero di costrutti di controllo
- Vasto numero di operatori
- Programmazione modulare (scomposizione in funzioni)
- Preprocessing

La struttura generale di un programma C è la seguente:

```
/* direttive al preprocessore */
#include <file1.h>
/*dichiarazione funzioni, strutture dati,
variabili globali*/
/* funzione main (programma principale) */
void main(void)
{
/*corpo del programma*/
}
```

Le direttive al preprocessore (`#include`) consentono l'inclusione nel programma di codice contenuto in altri files. In pratica servono a far sì che alcune funzionalità già codificate e raccolte in *librerie* facciano parte del programma in modo da poterle utilizzare. Le funzioni di ingresso/uscita `printf` e `scanf` che vediamo nel paragrafo 2.3 ne sono un esempio.

La definizione di funzioni permette di scomporre il programma in più moduli. Richiameremo questo concetto nel paragrafo

La funzione `main` implementa la struttura principale del programma.

Le parentesi `/* ...*/` racchiudono *commenti*, cioè parti di testo che non fanno parte del programma, servono solo per documentare il programma al fine di renderlo più comprensibile, e che vengono completamente trascurate dal compilatore, in modo che non hanno alcun effetto sull'esecuzione del programma.

2.1 Principali tipi di dati in C

Il C prevede i seguenti tipi di dati *predefiniti*:

Tipi di dati interi

- *char* (1 byte), definisce anche il tipo carattere;
- *int* (2 byte, nei computer a 16-bit; 4 byte, in quelli a 32-bit);

Tipi di dati reali

- *float* (4 byte);
- *double* (8 byte);

I tipi interi e reali possono essere ulteriormente definiti tramite i qualificatori:

- qualificatori dimensionali *short*, *long*: si applicano al tipo *int* e definiscono rispettivamente un dato la cui dimensione è \leq , oppure \geq a quella del tipo *int* (es: *short int* -> 2 byte);
- qualificatori aritmetici *signed*, *unsigned*: si applicano ai tipi *char* e *int* e definiscono l'aritmetica con o senza segno (i dati saranno trattati in forma complemento a due nel primo caso e come interi positivi nel secondo).

Type	Size
char, unsigned char, signed char	1 byte
short, unsigned short	2 bytes
int, unsigned int	4 bytes
long, unsigned long	4 bytes
float	4 bytes
double	8 bytes

Altre strutture dati:

Array: definiscono vettori i cui elementi appartengono tutti allo stesso tipo: ciascun elemento è contraddistinto da un indice:

```
int v[32]; /* definisce un vettore di 32 elementi interi, con indici che vanno da 0 a 31 */
```

Gli array possono anche avere più dimensioni, quindi posso definire delle matrici con dichiarazioni quali:

```
int mat[10][10];
```

che definisce una matrice 10 per 10 di interi.

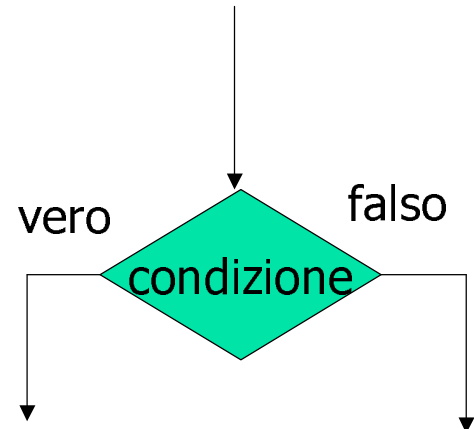
2.2. I costrutti di controllo del flusso

C presenta un numero ristretto di costrutti di controllo del flusso, cioè quei costrutti che permettono di modificare in modo condizionale il flusso di esecuzione di un programma, o che permettono di ripetere parti del programma; riportiamo di seguito i due costrutti più comuni (l'istruzione decisionale *if..else* e il ciclo *for*) con il corrispondente diagramma di flusso, al fine di presentare dei semplici esempi di programmi C.

Istruzione decisionale *if...else*

Sintassi:

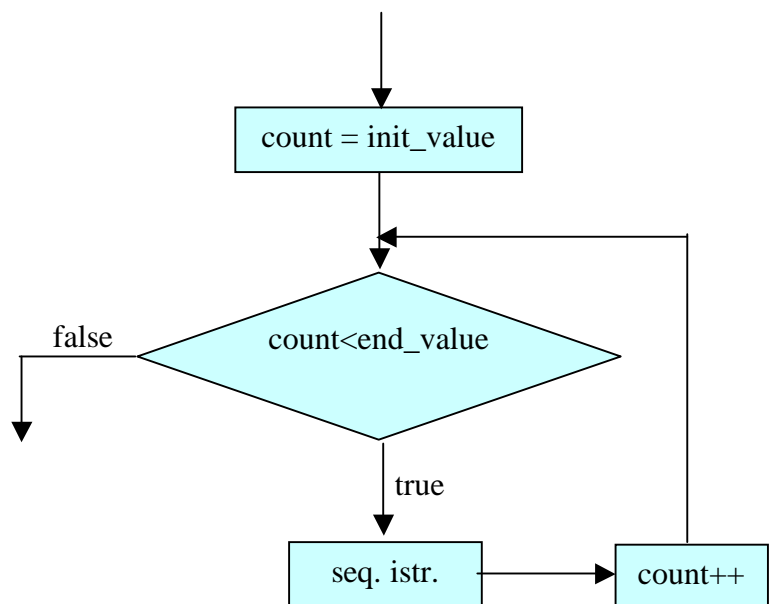
```
if ( <boolean expression> )
{
    < sequenza istruzioni >
}
else
{
    < sequenza istruzioni >
}
```



Ciclo *for*

Sintassi:

```
for(count=init_value; count<end_value; count++)
{
    < sequenza istruzioni >
}
```



2.3 Due funzioni fondamentali: printf, scanf

Altro elemento importante per poter scrivere un qualsiasi programma C è la capacità di comunicare con l'esterno, altrimenti il programma è del tutto inutile. Per questo introduciamo subito due funzioni C predefinite che permettono la lettura di valori da tastiera e la scrittura di valori sul monitor.

Printf

Visualizza sul dispositivo stdout (il display) messaggi e contenuto di variabili

Formato:

```
printf(< stringa di formato >, < argomenti >)
```

< stringa di formato >: messaggio che deve essere visualizzato, comprensivo dei riferimenti ai tipi di dati contenuti nelle variabili.

< argomenti >: variabili contenenti i dati da visualizzare.

Printf: esempio

(visualizzazione dei numeri primi compresi in 1..10)

```
int a=1; int b=2; int c=3; int d=5; int e=7;
```

```
printf("Numeri primi da 1 a 10: %d %d %d %d %d \n",a,b,c,d,e);
```

Messaggio visualizzato:

```
Numeri primi da 1 a 10: 1 2 3 5 7
```

“\n” è il carattere newline (ritorno a capo);

L'indicatore %d indica che nel testo deve essere inserito il valore intero corrispondente, per ordine, nella lista degli argomenti.

Altri indicatori di formato:

%d -> visualizzare un valore intero %c -> visualizzare un carattere (char)

%f -> visualizzare un valore reale (float)

%.< n >f -> visualizza un reale con le prime n cifre decimali dopo la virgola

Esempio: %.4f significa reale con 4 cifre decimale

Scanf

Serve per inserire dati in input tramite il dispositivo stdin (la tastiera) in opportune variabili

Formato:

```
scanf(< stringa di formato >, < argomenti >)
```

< stringa di formato >: indica il formato in cui saranno inseriti i valori

< argomenti >: variabili utilizzate per contenere i valori letti da tastiera; tutti gli argomenti devono essere preceduti dal carattere & (vedi paragrafo 2.9).

Scanf: esempio

(calcolo dell'area di un rettangolo)

```
float a,b,area;
```

```
printf("Inserire base e altezza: ");
```

```
scanf("%f %f", &a, &b); // si noti lo spazio che separa i due %f
```

```
area=(a*b);
```

```
printf("\nArea = %.4f", area);
```

Messaggio visualizzato:

```
Inserire base e altezza: 2 5
```

```
Area = 10.0000
```

i numeri devono essere separati dallo spazio!

2.4 Processo di compilazione

Come abbiamo già detto, il programma C non viene eseguito direttamente dal calcolatore, ma occorre tradurlo in un programma in linguaggio macchina, che possa quindi essere eseguito. Di ciò si occupa il compilatore, un programma che, preso un programma C, cioè un file con estensione `.c`, lo traduce in un cosiddetto *programma oggetto*, contenuto in un file con estensione `.obj`, che è essenzialmente un eseguibile in linguaggio macchina ancora da completare. Il completamento è compito del *linker* (collegatore), che collega il codice oggetto con le librerie di sistema necessarie per l'esecuzione del programma (si pensi alle funzioni di libreria `scanf` e `printf`) e produce l'eseguibile, cioè un file con estensione `.exe`, che può venir caricato in memoria e mandato in esecuzione dal Loader.

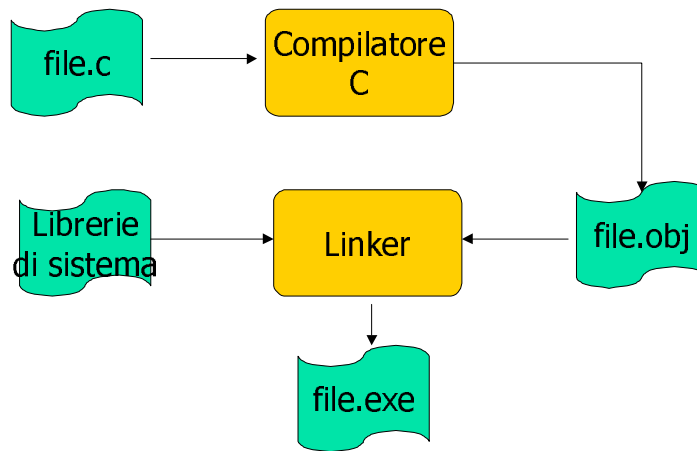


Fig. 3 Il processo di compilazione

Si noti che il compilatore, come pure il linker e il loader sono essi stessi dei programmi che vengono caricati in memoria e mandati in esecuzione. Quindi anche per scrivere questi programmi è stato usato un linguaggio di programmazione, forse proprio lo stesso C, nel qual caso sarà stato usato un (altro) compilatore per la loro compilazione. Evidentemente, in passato qualche compilatore deve essere stato programmato in linguaggio macchina...

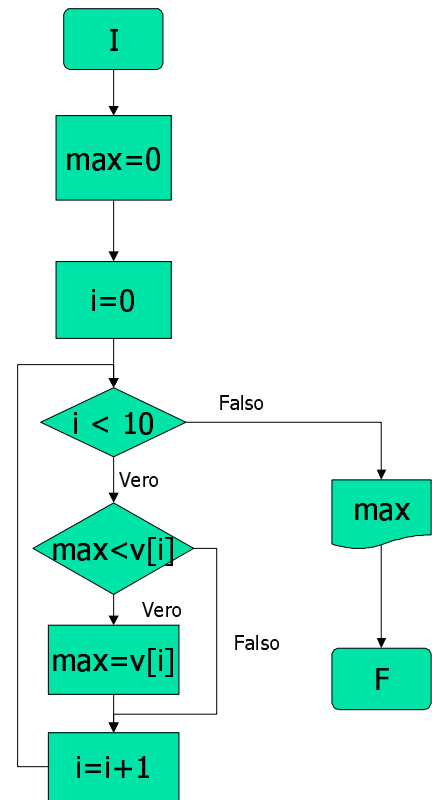
2.5 Esempi di programmazione

Ricerca del massimo elemento in un vettore

$V = \{13, 124, 324, 12, 214, 24, 325, 6543, -32, -2\}$

(10 elementi)

Algoritmo di scansione sequenziale: si controllano tutti gli elementi del vettore; se un elemento è maggiore del max trovato fino a quel momento, si aggiorna il max; si prosegue fino alla fine del vettore.



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

```
main(void)
```

```
{
```

```
    int v[10] = {13, 124, 324, 12, 214, 24, 325, 6543, -32, -2};
```

```
    int max, i, pos;
```

```
    printf("Ricerca elemento massimo in un vettore\n");
```

```
    max = 0;
```

```
    pos = 0;
```

```
    for (i = 0; i < 10; i++)
```

```
    {
```

```
        if (v[i] > max)
```

```
        {
```

```
            max = v[i];
```

```
            pos = i;
```

```
        }
```

```
    }
```

```
    printf("Valore massimo %d, posizione %d\n", max, pos);
```

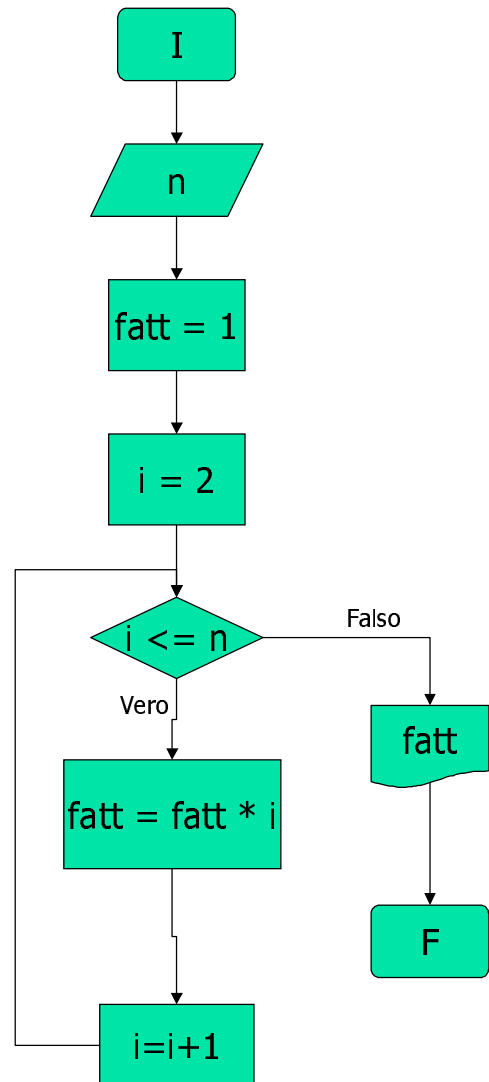
```
}
```

Fattoriale di un numero n

L'algoritmo si basa sulla

proprietà del fattoriale: $n! = n \cdot (n-1)!$

Con un ciclo si scorrono tutti i numeri da 1 a n ,
ricalcolando il fattoriale ogni volta
secondo la formula precedente.



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

```
int i,n;  
unsigned long fattoriale;
```

```
main(void)  
{
```

```
    printf("Numero: ");
```

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

```
    fattoriale = 1;
```

```
    for(i = 2; i<=n; i++)
```

```
        fattoriale *= i; //equivale a
```

```
    printf("%d! = %ld\n", n, fattoriale);
```

```
}
```

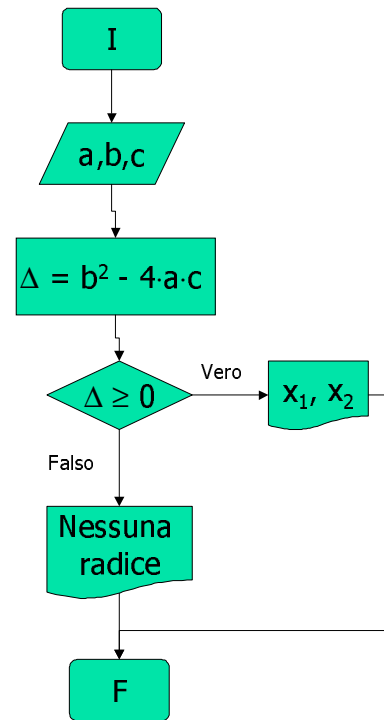
```
    fattoriale = fattoriale * i;
```

Risoluzione di un'equazione di 2° grado: $ax^2 + bx + c = 0$

- Si inseriscono i coefficienti a,b,c ;
- Si calcola $D = b^2 - 4 \times a \times c$;
 - $D \geq 0 \rightarrow$ 2 radici reali
($D = 0 \rightarrow$ radici coincidenti);
 - $D < 0 \rightarrow$ nessuna radice reale ;

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

```
main(void)
{
    float A, B, C, delta, X1, X2;
    printf("Calcolo delle radici dell'equazione di secondo grado: \n");
    printf("      ax^2 + bx + c = 0\n");
    //Inserisci i valori dei parametri a, b, c
    printf("inserisci i parametri a, b, c, nell'ordine: \n");
    scanf("%f %f %f", &A, &B, &C);
    printf("Radici dell'equazione con coeff. a = %f  b = %f  c = %f\n",A,B,C);
    if (A != 0) /*caso di equazione non degenera*/
    {
        /*calcola e visualizza le radici di ax^2+bx+c = 0*/
        delta = B*B - 4*A*C; /*calcolo del delta*/
        if (delta >= 0)
        {
            X1 = (-B - sqrt(delta))/(2*A);
            X2 = (-B + sqrt(delta))/(2*A);
            printf("Due radici reali: x1 = %f, x2 = %f\n", X1, X2);
        }
        else
        {
            printf("Nessuna radice reale!\n");
        }
    }
    else
        if (B != 0)
        {
            /*Caso degenera*/
            /*coeff del primo termine di grado diverso da zero*/
            X1 = - C / B;
            printf("Una sola radice: x1 = %f\n", X1);
        }
        else
        {
            printf("Nessuna soluzione!\n");
        }
}
```



2.6. I puntatori

Abbiamo detto come il concetto di variabile sia quello di un contenitore di valori, a cui viene associato un nome che lo identifica univocamente, e che viene realizzato mediante una locazione di memoria, associando quindi anche un indirizzo alla variabile. Finora, negli esempi di programmazione visti, abbiamo usato le variabili solo mediante il loro nome, ma in C è possibile rendere visibile ed utilizzare il loro indirizzo.

Introduciamo il concetto di *puntatore* a una variabile. Questo termine indica l'indirizzo della variabile: la dichiarazione:

```
typedef TipoDato *TipoPuntatore
```

definisce il tipo denominato TipoPuntatore come il tipo dei puntatori (indirizzi) a celle di memoria contenenti valori del tipo TipoDato. (typedef è una parola chiave riservata che introduce una definizione di un nuovo tipo, un cosiddetto *tipo definito da utente*, per distinguerlo dai *tipi predefiniti*.)

Quindi il valore di una variabile P di tipo TipoPuntatore è l'indirizzo di un'altra variabile, di tipo TipoDato. Questa raffigurazione è convenientemente rappresentata in modo grafico come in figura 4.

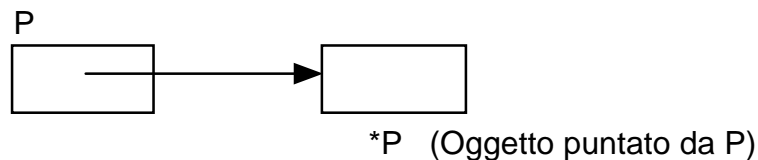


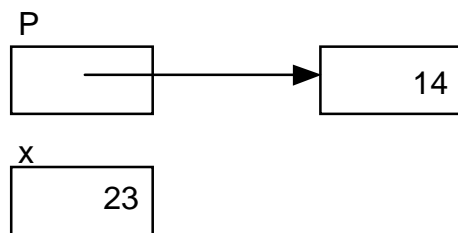
Figura 4 Rappresentazione grafica di un puntatore

L'accesso a una variabile "puntata" dal puntatore P avviene mediante l'operatore di *dereferenziazione*, indicato dal simbolo * che precede l'identificatore del puntatore. Perciò, *P indica la cella di memoria il cui indirizzo è contenuto in P.

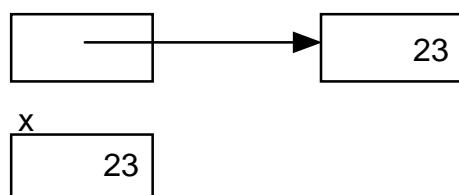
Assumendo che TipoDato sia il tipo int, e che abbia dichiarato le variabili:

```
TipoPuntatore P;  
TipoDato x;
```

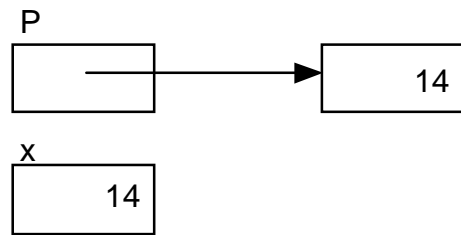
Se sono nella situazione:



ed eseguo il comando: `*P = x;`
ottengo la situazione:



Se invece nella situazione di partenza eseguo il comando: `x = *P;`
ottengo la situazione:



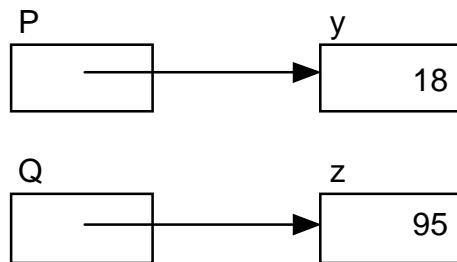
E' possibile ottenere l'indirizzo di una variabile con l'operatore `&`; pertanto, supponendo le dichiarazioni:

```
TipoPuntatore P,Q;  
TipoDato y,z;
```

le seguenti istruzioni, eseguite quando, ad esempio, `y` vale 18 e `z` vale 95 :

```
P = &y;  
Q = &z;
```

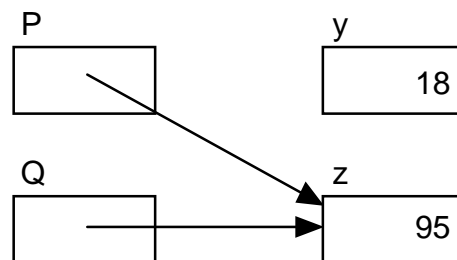
generano la situazione:



se ora eseguiamo il comando:

```
P = Q
```

otteniamo:



2.7. Altre strutture di controllo

Abbiamo utilizzato nei precedenti esempi il costrutto `for` per potere costruire algoritmi ciclici. Il costrutto `for`, nella sua forma più generale, ha la seguente *sintassi*:

(per *sintassi* si intende come deve essere scritto per essere corretto grammaticalmente, ma la *sintassi* non dice se un costrutto del linguaggio di programmazione è corretto da un punto di vista *semantico*, cioè del suo significato)

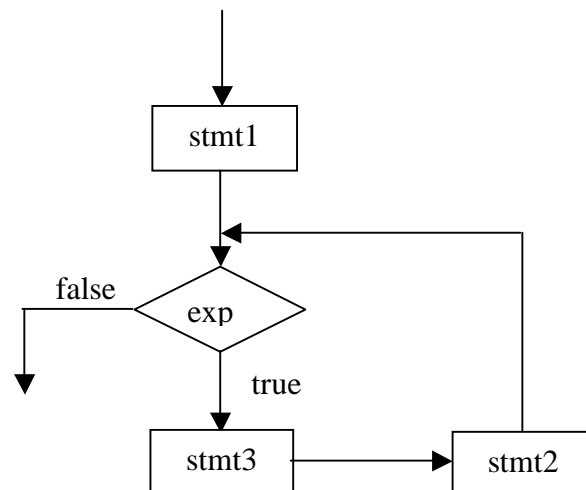
```
for (stmt1; exp; stmt2)
    stmt3;
```

dove:

`for` è una parola chiave riservata, cioè, insieme a `if`, `else`, `while`, `do` non può essere utilizzata per altri scopi, ad esempio come nome di una variabile.

`stmt1`, `stmt2`, `stmt3` sono dei comandi, in particolare `stmt1` è detto *inizializzazione*, `stmt2` è detto *legge di incremento* e `stmt3` è il comando che deve essere ripetuto. Si noti che tutti questi comandi possono essere comandi qualsiasi, incluso comandi composti, cioè *blocchi* in cui più comandi sono racchiusi tra parentesi graffe:

`exp` è un'espressione booleana di controllo del ciclo, per cui il ciclo si ripete per tutto il tempo che questa espressione resta vera



Si può notare come si sia utilizzato il linguaggio dei diagrammi di flusso per dare significato (semantica) a un costrutto di cui avevamo dato solo la sintassi.

La sintassi del ciclo `for` data in precedenza introduce una chiara distinzione tra espressione e comando:

Un'espressione è un metodo di calcolo in cui operatori elementari vengono combinati per costruire un'espressione che comunque *ritorna un risultato* di un certo tipo.

Un comando è invece un metodo di calcolo che *modifica delle variabili* (si dice anche *lavora per effetti laterali*) senza ritornare un risultato.

Un comando si *esegue*, un'espressione si *valuta*.

I comandi e le espressioni sono in genere strutture sintattiche ben distinte nei linguaggi di programmazione. In realtà, in C viene adottata una visione in cui le espressioni sono anche comandi e i comandi sono espressioni, per cui la loro valutazione/esecuzione può sia modificare delle variabili che ritornare un risultato.

Noi però continueremo a parlare di C come se la distinzione fosse mantenuta nel linguaggio.

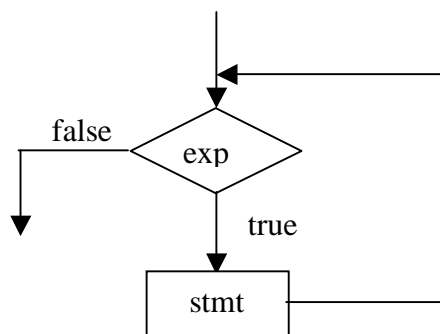
In C, come in altri linguaggi di programmazione ad alto livello, è possibile utilizzare altri costrutti più generici per realizzare cicli. Il `for`, infatti, pur essendo totalmente generico, come abbiamo visto poco sopra, è legato all'uso convenzionale che se ne fa per la scansione di array o di strutture simili. I costrutti che permettono di realizzare altre forme di ciclo sono:

while

sintassi:

```
while (exp)
    stmt;
```

semantica:

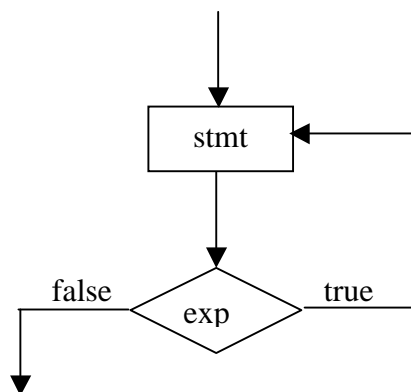


do while

sintassi:

```
do stmt;
while (exp)
```

semantica:



2.8. Sottoprogrammi

Gli esempi di programmi che abbiamo visto finora consistono semplicemente di un programma principale. Nella programmazione di sistemi anche appena più complessi è opportuno strutturare il

programma in *sottoprogrammi*, ognuno dedicato alla realizzazione di un particolare operazione del sistema.

Riprendiamo l'esempio, visto all'inizio del corso, dell'operazione che sposta una finestra su uno sfondo. L'operazione può essere realizzata con il seguente segmento di programma C:

```
int video[768][1024];
int sfondo[768][1024];
int finestra[230][400];
int i, j, x0, y0, x1, y1;
for(i=0; i<768; i++)
    for(j=0; j<1024; i++)
        video[i,j] = sfondo[i,j];
for(i=0; i<230; i++)
    for(j=0; j<400; i++)
        video[i+y1,j+x1] = finestra[i,j];
```

Sarebbe più utile, però, avere a disposizione un'operazione che definisco una volta per tutte e che posso riutilizzare quante volte voglio nel corso del programma, secondo quel principio per cui operazioni complesse si ottengono come composizione di operazioni più semplici, principio che abbiamo già incontrato più volte. Per far questo posso definire l'operazione di spostamento come un sottoprogramma:

```
void spostamento (int video[768][1024],
                  int sfondo[768][1024],
                  int finestra[230][400],
                  int x0, y0);
{
    int i,j;

    for(i=0; i<768; i++)
        for(j=0; j<1024; i++)
            video[i,j] = sfondo[i,j];
    for(i=0; i<230; i++)
        for(j=0; j<400; i++)
            video[i+y1,j+x1] = finestra[i,j];
}
```

Si noti come il sottoprogramma sia costituito di due parti, la prima parte detta *intestazione* dove oltre al nome del sottoprogramma si definiscono i parametri del sottoprogramma, cioè quello di cui il sottoprogramma ha bisogno per lavorare; la seconda parte detta *corpo* definisce invece l'algoritmo che calcola l'operazione.

L'operazione di spostamento è una *procedura*, in quanto esegue operazioni senza ritornare un risultato; si dice che *lavora per effetti laterali*, in quanto modifica delle variabili senza ritornare un risultato.

Se invece voglio scrivere un sottoprogramma che calcola l'operazione di fattoriale, posso scrivere una *funzione*, che ritorna un valore intero, e che non ha effetti laterali, in quanto il suo unico effetto è quello di ritornare il risultato, attraverso il comando *return*:

```
unsigned long fattoriale(int n);
{
    int i, unsigned long fatt;
    fatt = 1;
    for(i = 2; i<=n; i++)
        fatt = fatt *i;
    return fatt;
}
```

Le funzioni e le procedure si distinguono anche per come vengono *chiamate*:

se in un programma voglio utilizzare la procedura di spostamento di una finestra, dato un video e uno sfondo, alle coordinate (15,25) basterà scrivere l'istruzione:

```
spostamento(video, sfondo, finestra,15,25)
```

che è un comando e che si dice *chiamata* alla procedura.

Perché questa chiamata vada a buon fine (cioè il compilatore non rilevi un errore) occorre:

- che nel programma sia presente la definizione della procedura spostamento, data in precedenza,
- che nel programma siano state dichiarate le variabili finestra, video e sfondo, in modo conforme a quanto richiesto dalla procedura, cioè come array dalle dimensioni indicate,
- che gli identificatori delle variabili finestra, video e sfondo, e quello della procedura spostamento siano *visibili* nel punto del programma dove appare questa chiamata (vedi paragrafo. 2.9).

La funzione fattoriale si può chiamare con la seguente istruzione:

```
f = fattoriale(n);
```

purchè f sia una variabile dichiarata come unsigned long, n una variabile dichiarata come int, e siano rispettate le condizioni di visibilità degli identificatori.

L'istruzione ha l'effetto di assegnare il valore calcolato dalla funzione alla variabile f. La chiamata alla funzione fattoriale in questo caso è una espressione.

In C, sia le funzioni che le procedure (cioè tutti i sottoprogrammi) prendono in realtà il nome di *funzioni*, e si distinguono solo dal tipo del risultato: *void* indica che non c'è risultato e che la funzione lavora per effetti laterali. Questo implica che anche quella che avremmo considerato una funzione, cioè un sottoprogramma che ritorna un risultato, può lavorare anche per effetti laterali, quindi si possono scrivere funzioni che hanno come effetto sia quello di modificare variabili, sia quello di ritornare un risultato; questo in ossequio alla filosofia adottata in C in cui non vi è distinzione tra espressioni e comandi. Questo è un indice della grande flessibilità del linguaggio, che talvolta rende meno comprensibile (e quindi più propenso a contenere errori) un programma, in quanto, leggendo il testo di una funzione C, non sempre ne sono immediatamente identificabili gli effetti, ma questi devono essere pazientemente ricercati nel testo stesso.

I valori che compaiono tra parentesi dopo il nome della funzione nella intestazione della funzione vengono detti *parametri formali*, mentre quelli che appaiono tra parentesi nella chiamata si dicono *parametri attuali*. Perché una chiamata di funzione vada a buon fine ci deve essere compatibilità tra parametri attuali e formali che devono corrispondere in numero, ordine e tipo. Al momento della chiamata avviene il passaggio dei parametri alla funzione chiamata mediante la scrittura dei valori dei parametri attuali nei corrispondenti parametri formali, che a tutti gli effetti sono variabili locali alla funzione. Questa forma di passaggio di parametri viene detta *passaggio per valore* o per copia.

2.9. Regole di visibilità degli identificatori

Il campo di visibilità (detto in inglese *scope*) di un identificatore (cioè di un nome, sia esso di costante, variabile, funzione o tipo), si definisce come la zona di programma dove si può usare l'identificatore stesso come riferimento allo stesso oggetto. La definizione del campo di visibilità prende a riferimento il blocco circostante alla definizione. Per blocco si intende la zona del programma racchiusa tra parentesi graffe.

In particolare il campo di visibilità va dal punto di definizione dell'identificatore (dichiarazione di variabile o costante, definizione di funzione o tipo) fino alla parentesi graffa chiusa che chiude l'ultima parentesi graffa aperta precedente la definizione. Questa regola implica le seguenti:

- gli identificatori possono essere ridefiniti: in questo caso vale la definizione del blocco più interno che racchiude il punto di utilizzo dell'identificatore;
- gli identificatori definiti in testa al programma sono globali cioè visibili da tutto il programma, e possono essere utilizzati in tutto il programma;
- gli identificatori definiti all'interno di un blocco sono detti *locali*, e non sono utilizzabili dall'esterno del blocco.

In particolare, dichiarazioni di variabili che appaiono in testa al programma definiscono variabili globali, mentre quelle che appaiono all'interno di funzioni definiscono variabili locali.

Ad una variabile globale si può assegnare un valore in qualsiasi punto del programma, quindi anche all'interno di una funzione; mentre ad una variabile locale si può assegnare un valore, o si può leggerne il valore, solo all'interno della funzione che la definisce. Quindi, una scrittura di un valore in una variabile locale non sopravvive alla terminazione della funzione, mentre la scrittura di un valore in una variabile globale sopravvive alla terminazione della funzione.

La scrittura in una variabile globale corrisponde quindi ad un *effetto laterale* della funzione, e può essere utilizzata per ottenerne risultati, soprattutto quando la funzione C operi in realtà come una procedura.

Consideriamo un frammento di programma come esempio:

```
...
int x, y;
...
boolean ricerca(vet v, key k)
{
    int x,z;
    ...
}
void main(void)
{
    int z;
    ...
}
```

Nella figura 5 la struttura a blocchi del programma viene rappresentata con delle scatole, che definiscono esattamente lo scope degli identificatori in esse definiti; il campo di variabilità dei vari

identificatori è inoltre raffigurato dall'estensione delle frecce; si noti come la definizione più interna della variabile x "nasconde" quella più esterna. Le due variabili x, come le due variabili z, sono variabili distinte, benché condividano il nome e il tipo. Le variabili x e y dichiarate esternamente sono variabili globali (si noti però come x non possa essere utilizzata dentro la funzione ricerca, perché nascosta dalla definizione locale di x). Gli identificatori v e k, parametri formali della funzione ricerca, sono a tutti gli effetti delle variabili locali alla funzione ricerca, come se fossero dichiarate all'interno delle parentesi graffe. Invece, l'identificatore ricerca, che nel programma originale si trovava sulla stessa riga di v e k, ha un campo di variabilità esterno al blocco che definisce, in modo che la funzione possa essere chiamata da ogni punto in cui si vede il suo identificatore.

Le regole introdotte permettono anche che una chiamata ad una funzione sia all'interno del suo blocco. si parla in questo caso di chiamata ricorsiva (una funzione che chiama se stessa): questa è una importante possibilità, che però tralasciamo nell'ambito di questo corso.

Da quanto detto risulta come sia importante la struttura a blocchi in un programma C. Per questo è molto utile adottare il metodo di scrittura di programmi detto *indentamento*, in cui all'entrare in un nuovo blocco il testo del programma viene scritto spostato a destra di qualche carattere bianco, in modo da evidenziare la struttura *annidata* dei blocchi. Questo modo di scrivere il programma è del tutto ininfluenza sul significato del programma e viene trascurato dal compilatore, mentre è utilissimo per la comprensione umana del programma.

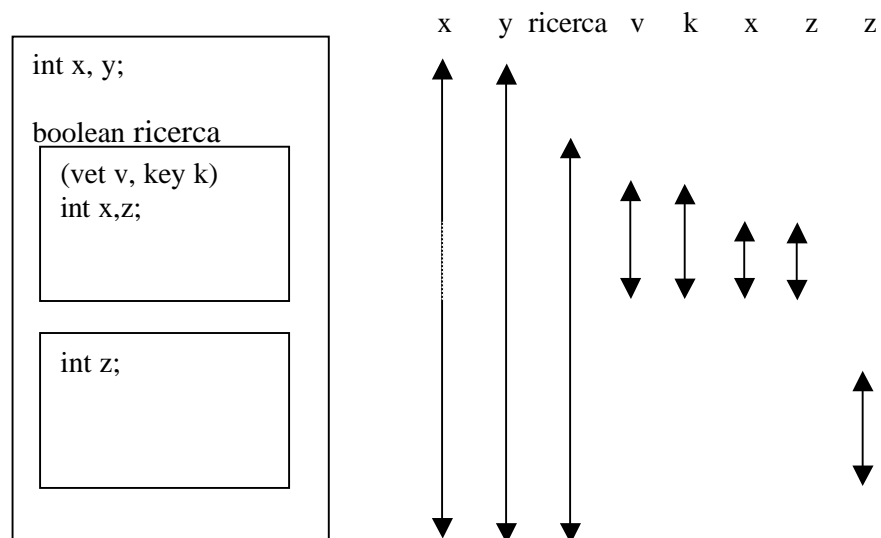


Fig. 5 Struttura a blocchi di un programma e campo di visibilità degli identificatori

Quanto abbiamo detto sui puntatori ci suggerisce inoltre un altro modo di ottenere risultati da una funzione C operante come una procedura; spesso ci si riferisce a questa tecnica come *passaggio per riferimento* o per indirizzo.

In questa tecnica si passa come parametro ad una funzione C l'indirizzo di una variabile, inteso come valore di un tipo puntatore appropriato. Conoscendo questo indirizzo, con l'operatore di dereferenziazione la funzione potrà leggere e scrivere nella variabile passata per indirizzo, e queste modifiche sopravviveranno alla terminazione della procedura.

Consideriamo come esempio la seguente funzione:

```
void ricerca(vet v, key k, boolean *ris);
{
```

```

...
if (v[i]=k) *ris = TRUE;
...
}

```

Una chiamata di questa funzione sarà del tipo:

```

boolean trovato;
ricerca(Anagrafe, 'Carlo Rossi', &trovato);
if (trovato) ...
...

```

Possiamo ora comprendere perché nella printf il parametro attuale da cui scrivere il valore su video viene passato per valore, mentre quando chiamiamo la scanf dobbiamo passare l'indirizzo della variabile in cui viene scritto il valore letto da tastiera.

Infine, in C gli array vengono considerati non esattamente come una variabile composta che contiene tutti i valori del vettore, ma piuttosto come un puntatore ad un'area di memoria dove sono memorizzati tutti i valori. Pertanto, quando si passa (per valore) un array a una funzione, in realtà si passa per valore questo indirizzo e quindi si passano per riferimento le locazioni di memoria contenenti i valori dell'array.. In questo modo la funzione *spostamento* che abbiamo presentato poco sopra è corretta, in quanto essendo l'array video passato per riferimento, ogni modifica alla variabile video fatta internamente alla funzione, si ritrova una volta che si esce dalla funzione, perché non vi è stata copia dei valori dell'array in un parametro formale. Questo meccanismo, che è una particolarità del C, evita tra l'altro la copia di strutture di grandi dimensioni, come nel caso della funzione spostamento, che sarebbero onerose dal punto di vista del tempo di esecuzione della operazione.

2.10 Alcuni esempi di algoritmi e cenni alla complessità computazionale.

Algoritmi di ricerca in un vettore: ricerca esaustiva

La seguente funzione C ricerca un elemento in un vettore, vale a dire torna il valore vero se l'elemento è presente nel vettore, e un valore falso altrimenti. Per far questo esegue un algoritmo di ricerca esaustiva, cioè scandisce il vettore fino a quando o si è trovato l'elemento, o si è finito di scandire il vettore.

Supponiamo che `vett` sia un tipo array definito come:

```
typedef elem vett[dim]
```

dove `dim` è un numero (non una variabile!), e `elem` è un tipo qualsiasi.

La funzione prende come parametro anche la lunghezza del vettore; in questo modo possiamo descrivere l'algoritmo in modo generale.

```

boolean ricerca (vett z, int n, elem x);
{
    int i; boolean flag;
    flag = FALSE;
    for (i=0; i<n; i++)
        if (z[i] == x)    flag =  TRUE;
    return flag;
}

```

Si noti che la funzione esegue al più tante iterazioni del ciclo while quanti sono gli elementi dell'array. Se questi sono n, il tempo occorrente per eseguire il calcolo è quindi proporzionale a n. Si dice che questa funzione ha *complessità computazionale lineare*.

D'altra parte, è inutile continuare ad eseguire il ciclo quando si fosse già trovato l'elemento; la seguente funzione introduce una nuova condizione nell'while, che permette di uscire anticipatamente dal ciclo quando questo si verifica (le operazioni booleane in C si indicano con i simboli: ! per il NOT, && per l'AND, || per l'OR).

```

boolean ricerca (vett z, int n, elem x);
{
    int i; boolean flag;
    i=0;
    flag = FALSE;
    while (i<n && !flag)
        {
            if (z[i] == x)    flag =  TRUE;
            i++
        }
    return flag;
}

```

Senza altro questo algoritmo è più vantaggioso del precedente, perché si sono eliminati dei passi inutili. Ma possiamo considerare che ora il tempo di esecuzione della funzione dipende dall'elemento che cerchiamo: se l'elemento non è presente nel vettore, si eseguono n passi; se è presente, si eseguono k passi, dove k è la sua posizione nel vettore. In media, k varrà n/2. Possiamo quindi dire che il tempo occorrente è comunque ancora proporzionale a n, e che questa funzione ha ancora *complessità computazionale lineare*.

Ricerca binaria o dicotomica

Un significativo miglioramento dell'efficienza si ha con l'algoritmo di *ricerca binaria* o *dicotomica*. Questo algoritmo presuppone che il vettore sia ordinato (in modo crescente) e si basa sul seguente principio: confronto l'elemento cercato con l'elemento di mezzo del vettore. Se non è uguale, se è minore posso andarlo a cercare nel primo semivettore, se è maggiore nel secondo semivettore. La ricerca nei semivettori adotta lo stesso procedimento, fino a quando si trovi l'elemento o si arrivi a considerare un semivettore di lunghezza unitaria

```

boolean ricerca (vett z, int n, elem x);
{
  int k, m;
  k=0; m=n;
  int i; boolean flag;
  flag = FALSE;
do
  {
    i = (m+n) / 2

    {
      if (z[i] == x)    flag =  TRUE;
      if (z[i] < x)
        k = i;
      else
        m = i;
    }
  while (k<m && ! flag)
  return flag;
}

```

Questa funzione esegue un numero di iterazioni del ciclo al più uguale al logaritmo di n in base 2. Infatti il vettore viene diviso per 2^p volte, fino a quando non diventa $n/2^p = 1$, cioè $n = 2^p$, e quindi $p = \log_2 n$.

Il tempo occorrente per il calcolo è quindi al più proporzionale al logaritmo di n, e si dice che questa funzione ha *complessità computazionale logaritmica*. Per capire quanto sia più vantaggioso questo algoritmo di ricerca rispetto alla ricerca esaustiva basta fare un semplice esercizio in cui si consideri come cresce il tempo di esecuzione al crescere della dimensione del vettore, per valori della dimensione del vettore pari a 10, 100, 1000, 10000.

D'altra parte, questo algoritmo assume che gli elementi del vettore siano ordinati, quindi assume anche che il tipo *elem* degli elementi sia un tipo su cui è definito un ordinamento totale e una operazione di confronto (l'operazione <).

Lo strumento della complessità computazionale, di cui qui non diamo una definizione formale, è quindi utile a classificare gli algoritmi per la loro potenziale efficienza nel trattare problemi di grandi dimensioni. Un algoritmo che avesse una complessità computazionale esponenziale, cioè il cui tempo di esecuzione cresce come 2^n potrebbe essere utile solo per piccole dimensioni del problema, e completamente inutile da un punto di vista pratico per dimensioni appena maggiori, quand'anche si avesse a disposizione un calcolatore molto potente, cioè molto veloce nell'eseguire le istruzioni. Per capire quanto questo sia vero si consideri come esercizio quanto vale il tempo di 2^n microsecondi, per $n = 10, 100, 1000, 10000$.

Si noti infine che alla minore complessità computazionale della ricerca binaria fa riscontro una "maggiore complicazione" dell'algoritmo.

2.11 Algoritmi di ordinamento

Come ulteriore esempio di algoritmi diversi, con diversa complessità computazionale, che calcolano la stessa funzione, mostriamo alcuni algoritmi di ordinamento di un vettore.

Questi algoritmi condividono la definizione del tipo vettore:

```
typedef tipoelem vett[dim];
```

e la definizione di una procedura di scambio del valore di due variabili. Si noti come lo scambio richieda una variabile di appoggio, e come le variabili da scambiare siano passate alla procedura per indirizzo.

Procedura di scambio di due elementi di un vettore

```
void scambia (tipoelem *x, *y)
{
    tipoelem aux;
    aux=*x;
    *x=*y;
    *y=aux;
}
```

Notare, nelle procedure che seguono, che il vettore da ordinare non è, apparentemente, passato per indirizzo; si deve invece ricordare che i vettori sono trattati in C come puntatori al loro primo elemento, quindi sono sempre passati per indirizzo.

Selection_sort (ordinamento per selezione)

La procedura di ordinamento per selezione esegue, per ogni passo del ciclo esterno, un ciclo interno che sceglie l'elemento minimo e lo porta nella prima posizione, usando la procedura di scambio data sopra definita. Il prossimo passo del ciclo esterno si può limitare ad ordinare il resto del vettore, ripetendo lo stesso algoritmo. Ad ogni passo vengono quindi considerati vettori sempre più piccoli, fino ad arrivare a un vettore di lunghezza unitaria che è ordinato per definizione.

```
void selection_sort (vett z, int n);
{
    int i,j,imin;
    for (i=0; i<n-1; i++)
    {
        imin=i;
        for (j=i+1; j<n; j++)
            if (z[j]<z[imin])
                imin = j;
        scambia(&z[i], &z[imin]);
    }
}
```

Per valutare la complessità di questo algoritmo, notiamo che il ciclo interno compie $n-2$ passi la prima volta, $n-3$ la seconda, $n-4$ la terza, e così via. Quindi complessivamente il numero di passi è

pari a $\sum_{i=1, n-2} i = (n-2)*(n-1)/2 = n^2/2 - 3*n/2 - 1$:

Possiamo quindi dire, trascurando i termini di ordine inferiore, che il tempo occorrente per l'ordinamento è proporzionale a n^2 , e che questa funzione ha *complessità computazionale quadratica*.

Bubblesort (ordinamento a bolle)

Questo algoritmo opera scandendo il vettore a partire dal fondo; man mano che trova due elementi successivi in ordine scorretto, li scambia. In questo modo l'elemento minimo "galleggia" via via verso la prima posizione del vettore, dopo di che si può passare ad ordinare il resto dell'array. Il nome dell'algoritmo è ripreso dal fatto che gli elementi di valore minore tendono a risalire come le

bolle in una boccia d'acqua. Un vantaggio di questo algoritmo è che, mentre si scandisce il vettore dal basso verso l'alto, si può controllare se il vettore risulta già ordinato (se non si sono fatti scambi, vuol dire che il vettore è già ordinato). In questo caso, posso abbandonare la procedura.

```
void bubblesort (vett z, int n);
{
  int i,j;
  boolean fine;
  i = 0
  do
    {
      i++;
      fine = TRUE;
      for (j=n-1; j>=i; j--)
        if (z[j]<z[j-1])
          {
            scambia(&z[j], &z[j-1]);
            fine = FALSE;
          }
    }
  while (!fine && i!=n-1)
}
```

Si può vedere facilmente che le argomentazioni per il calcolo della complessità di questo algoritmo, sono le stesse dell'algoritmo di ordinamento per selezione, con la variante che esistono casi in cui l'algoritmo può terminare prematuramente, quando (porzioni de) il vettore risulti già ordinato. Possiamo però considerare che probabilisticamente questo sia un evento raro, e che questa funzione presenta mediamente una *complessità computazionale quadratica*.

Quicksort (ordinamento veloce)

Questo algoritmo si basa su un principio diverso, che mira, in analogia con l'algoritmo di ricerca binaria, ad abbattere la complessità computazionale suddividendo il vettore in due parti, in modo da ripetere l'ordinamento di sottovettori solo un numero di volte pari al logaritmo della dimensione.

L'algoritmo fissa dapprima un elemento detto *pivot*; attraverso una scansione simultanea dal primo elemento in poi e dall'ultimo all'indietro, e scambiando gli elementi eventualmente trovati non in ordine in questa scansione, suddivide il vettore nel sottovettore contenente tutti gli elementi minori del pivot, e in quello contenente gli elementi maggiori. In particolare, nella funzione che segue, viene scelto come pivot il primo elemento del vettore; la scansione viene effettuata mediante due indici, che si muovono dagli estremi del vettore in senso contrario fino a toccarsi.

Una volta compiuta la suddivisione nei due sottovettori, si passa ad applicare lo stesso algoritmo ai due sottovettori, fino a quando i sottovettori non diventino di dimensione unitaria. Per poter far questo, occorre avere a disposizione una lista in cui memorizzo le coppie di indice minimo e massimo dei sottovettori da ordinare. Quindi la funzione ripetutamente applicherà il procedimento visto sopra, a partire dall'intero vettore; questo procedimento inserirà nella lista gli indici di due sottovettori, a meno che i sottovettori non risultino di un elemento. Nei passi successivi si estrarranno dalla lista gli indici di uno dei vettori ancora da ordinare e si applicherà a questo il procedimento di suddivisione, finché la lista non contenga più coppie di indici.

Nella funzione che segue si è appunto supposto di avere realizzato una lista e di avere a disposizione delle funzioni che operano sulla lista (inserzione, estrazione, test che la lista sia vuota). Una tale struttura si può agevolmente realizzare con un vettore.

```

void quicksort (vett z, int n)
{
    int sin,des, i,j;
    tipoelem x;
    tipolista lis;          /*suppongo l'esistenza di un tipo lista
                             su due valori interi, con funzioni ins e estr.
                             Nella lista vengono memorizzate le coppie di indice
                             minimo e massimo dei sotto-vettori ancora da ordinare*/
    ins (&lis, 0, n-1);
    while (!listavuota(lis))
    {
        estrai (&lis, &sin,&des);
        i=sin;
        j=des+1;
        x=z[sin];
        while(i<j)
        {
            do j--;
            while(z[j]>x);
            do i++;
            while(z[i]<=x && i!=j);
            if (i<j)
                scambia(&z[i], &z[j]);
        }
        if (sin!=j)
            scambia(&z[sin], &z[j]);
        if (sin<j-1)
            ins (&lis, sin,j-1);
        if (des>i)
            ins (&lis, i,des);
    }
}

```

Per valutare la complessità di questo algoritmo, supponiamo innanzitutto che il vettore sia di dimensioni tali che la sua suddivisione successiva in sottovettori sia uniforme: in questo caso avremo che la prima suddivisione produce 2 vettori di $n/2$ elementi ciascuno, la seconda produce 4 vettori di 4 elementi ciascuno, ... la p -esima 2^p sottovettori di $n/2^p = 1$ elementi, quindi $p = \log_2 n$. La scansione di ciascuno dei sottovettori è lineare, quindi la prima suddivisione del vettore intero viene fatta in n passi, la seconda prevede due scansioni di $n/2$ passi ciascuna, in totale n passi, la terza quattro scansioni di $n/4$ passi, in totale ancora n passi, etc.... Il totale di passi fatti dall'algoritmo ammonterà pertanto a $n * \log_2 n$ e il tempo occorrente per l'ordinamento sarà proporzionale a $n \log n$. Quindi questa funzione ha *complessità computazionale semilogaritmica*, che è comunque molto migliore di quella quadratica degli altri algoritmi. Questo risultato, che giustifica il nome dell'algoritmo, si è ancora una volta ottenuto al prezzo di una "maggiore complicazione" dell'algoritmo.