CORSI ABILITANTI SPECIALI
L. 143/ter D.M. 85/2005
SIS TORINO

Appunti di
SVILUPPO SOFTWARE IN C

A cura di Domenico Valle

A. A. 2006/2007

Sommario

CAPITOLO 1

Introduzione

1.1 Il linguaggio C

Il linguaggio C nasce nel 1972 ad opera di Denis Ritchie. Discende dal linguaggio B, che a sua volta è una rielaborazione , fatta nel 1970 da Thompson, del linguaggio BCPL, creato da Martin Richards nel 1967. Tra le principali caratteristiche del linguaggio possiamo evidenziare che :

- può essere considerato un linguaggio a basso livello nel senso che permette una buona interazione con l'hardware;
- può essere anche considerato un linguaggio ad alto livello perché dotato delle strutture di controllo tipiche dei linguaggi di questo tipo;
- è facilmente trasportabile;
- è "povero" in quanto non gestisce direttamente molte operazioni, ad esempio l'input-output; si supplisce a tale inconveniente con le funzioni che costituiscono le "librerie" del C;
- uno dei principali difetti è la concisione che può rendere un programma C poco leggibile; è opportuno allora corredare il programma di commenti e di una adeguata documentazione.

1.2 Struttura di un programma C

Un **programma sorgente** C è un insieme qualunque di direttive, dichiarazioni, definizioni ed istruzioni. Un programma C può essere composto da più **file sorgenti**, che vengono compilati separatamente e successivamente linkati insieme per ottenere un unico file eseguibile.

- **direttiva** : comando che impone al **preprocessore** C di compiere determinate azioni sul testo del programma prima che esso venga compilato;
- **dichiarazione di variabile**:associazione tra il nome e gli attributi di una variabile, di una funzione, di un tipo;
- **definizione di variabile** : stessa associazione di una dichiarazione ma in più viene riservata della memoria per la variabile definita; tutte le definizioni sono dichiarazioni ma non viceversa;
- **dichiarazione di funzione**: prototipo della funzione;
- **definizione di funzione** : intestazione + corpo della funzione; la definizione di funzione deve essere fatta sempre al livello esterno.

Nella sua forma più semplice un programma C può presentarsi nel seguente modo :

- Direttive al pre-processore
- prototipi delle funzioni
- definizione delle variabili globali
- main()

```
{
  .....
  .....
}
```

- definizione della funzione_1
- .....
- definizione della funzione_n

ogni **istruzione** deve essere **chiusa** dal **;**
i commenti devono essere racchiusi tra **/*** **e** ***/** ; in alternativa, per commenti che non si estendono per più di una riga, il commento deve essere preceduto da //.

Esempio :

```
/* programma somma */
#include <stdio.h>                          // direttiva al preprocessore
void stampa (void);                         // dichiarazione (prototipo) della funzione stampa
int a,b;                                    // definizione delle variabili globali a e b
main()
        {
          scanf("%d",&a);                   // chiamata della funzione di libreria scanf
          scanf("%d",&b);                   // chiamata della funzione di libreria scanf
          stampa();                         // chiamata della funzione stampa
        }
void stampa(void)                           // definizione della funzione stampa
        {
          int c;                            // definizione della variabile locale c
          c=a+b;                            // istruzione
          printf ("%d",c);                  // chiamata della funzione di libreria printf
          return;
        }
```

CAPITOLO 2

Dati e operatori

2.1 Identificatori

Si chiamano **identificatori** i nomi utilizzati dal programmatore per "identificare", appunto, un determinato oggetto (ad esempio, sono identificatori: i nomi di variabile, i nomi di funzione).

Un **identificatore** è costituito da una sequenza di caratteri (lettere maiuscole o minuscole, cifre, carattere di sottolineatura (underscore : _)); può iniziare con una lettera o underscore (sebbene quest'ultima possibilità non è consigliabile) ma non con una cifra. Secondo l'ANSI C standard i caratteri maiuscoli e minuscoli sono caratteri diversi e la lunghezza massima di un identificatore è di 31 caratteri.

2.2 Tipi di dati

TIPO	ESTENSIONE(n. byte)	INTERVALLO VALORI
Char	1 byte	da -128 a +127
Int	2 byte	da -32768 a + 32767
Short [int]	2 byte	da -32768 a + 32767
long [int]	4 byte	da -2147483648 a +2147483647
Unsigned char	1 byte	da 0 a 255
Unsigned [int]	2 byte	da 0 a 65535
Unsigned short [int]	2 byte	da 0 a 65535
Unsigned long [int]	4 byte	da 0 a 4294967295
Float	4 byte	3.4E +/-38 (7 cifre) (*)
Double	8 byte	1.7E +/-308 (15 cifre) (*)
long double	10 byte	1.2E +/-4932 (19 cifre) (*)
Puntatori	2 byte	

(*) Rappresentazione floating point nella notazione ANSI/IEEE 754 standard; in tale notazione i valori minimi sono, in realtà : 1.18E-38 (float), 2.23E-308 (double), 3.4E-4932 (long double).

Per evitare risultati errati, bisogna tener presente che, in una espressione in cui sono presenti operandi di tipo diverso, il C opera le cosiddette **conversioni implicite,** per cui l'operando di tipo inferiore viene convertito nel tipo dell' operando di livello superiore, secondo la seguente gerarchia (dal livello più alto al livello più basso) :

- long double
- double
- float
- unsigned long
- long
- unsigned int
- int

Inoltre :

- i tipi char e short vengono sempre convertiti in int;
- i tipi unsigned short e unsigned char vengono sempre convertiti in unsigned int;
- nell' operazione di assegnamento l'operando di destra viene sempre convertito nel tipo dell'operando di sinistra.

Esempi di definizione di variabile:

1. int x,y,z ; // x,y,z variabili intere
2. char a,b,c; // a,b,c variabili carattere
3. float m,n,q; // m,n,q variabili reali singola precisione
4. double e,f,g; // e,f,g variabili reali doppia precisione

È possibile inizializzare le variabili in sede di definizione; ad esempio :

1. char var1='c', var2='à; // var1 e var2 variabili carattere inizializzate con 'c' e 'à
2. int x=0 ; // x variabile intera inizializzata a 0

2.3 Operatori

DESCRIZIONE	OPERATORE	DESCRIZIONE	OPERATORE
Funzione	()	Scorrimento a sinistra	<<
Array	[]	Scorrimento a destra	>>
Indirizzamento struttura	- >	Minore di	<
Membro di struttura	.	Minore o uguale a	<=
Incremento Decremento	++ --	Maggiore di	>
Complemento a 1	~	Maggiore o uguale a	>=
NOT unario	!	Uguale	==
Indirizzo (unario)	&	Diverso	!=
Riferimento (contenuto puntato da) (unario)	*	AND bit a bit	&
Conversione di tipo	(tipo)	XOR bit a bit	^
Meno unario	-	OR bit a bit	|
Dimensioni in byte	sizeof	AND logico	&&
Moltiplicazione	*	OR logico	||
Divisione	/	Condizionale	?:
Resto	%	Assegnamento	= %= += -= *= /= >>= <<= &= ^= |=
Addizione	+	Virgola	,
Sottrazione	-		

Precedenze (in ordine decrescente di priorità) e associatività degli operatori

OPERATORE	ASSOCIATIVITÀ
() [] . ->	da sinistra
++ --	da destra
! - + ~ (tipo) * & sizeof (+,-,*,& unari)	da destra
* / % (* moltiplicazione)	da sinistra
+ - (+ addizione - sottrazione)	da sinistra
>> <<	da sinistra
< > <= >=	da sinistra
== !=	da sinistra
& (& AND bit a bit)	da sinistra
^	da sinistra
|	da sinistra
&&	da sinistra
||	da sinistra
?:	da destra
= += -= *= /= %= <<= >>= &= |= ^=	da destra
,	da sinistra

L'ordine con cui vengono valutate le espressioni dipende dalla **precedenza** degli operatori (vengono valutate prima le operazioni con operatori a priorità più alta) ; quando gli operatori hanno la stessa priorità è la **associatività** (modo in cui devono essere raggruppati gli operandi) a determinare l'ordine di esecuzione delle operazioni. Ad esempio :

1. nell'espressione 36 + 6 / 6 viene valutata prima la divisione (6 / 6) e dopo l'addizione, dato che la divisione (/) ha priorità più alta dell'addizione(+);
2. nell'espressione 64 / 8 * 2 dato che la divisione(/) e la moltiplicazione(*) hanno la stessa priorità, sarà l'associatività a determinare l'ordine della valutazione dell'espressione; in questo caso (associatività da sinistra verso destra) viene prima valutata la divisione e poi la moltiplicazione cioè : (64 / 8) * 2; nel caso si avesse un'associatività da destra verso sinistra l'espressione dovrebbe essere considerata come : 64 / (8 * 2);
3. nell'espressione a = b = c = d = 0, dato che per l'assegnamento vale l'associatività da destra verso sinistra. l'espressione dovrebbe essere considerata come : a = (b = (c = (d = 0))).

Esempi : x = y * z-- equivale a x = y * (z--)
x /= y * 3 + 5 / z++ equivale a x = x / ((y*3) + (5 / (z++)))

2.4 Operatori aritmetici

Sono

- **+ addizione**
- **- sottrazione**
- *** moltiplicazione**
- **/ divisione**
- **% modulo (**o **resto)**
- **- negazione (unario)**
- **+ segno + (unario)**
- **++ incremento unitario (unario)**

- **-- decremento unitario (unario)**

I primi 4 operatori effettuano le 4 operazioni aritmetiche elementari; l'operatore modulo (**%**) permette di ottenere il resto della divisione (intera) tra i suoi 2 operandi, che devono essere di tipo intero; la negazione (-) permette di ottenere l'opposto dell'operando; il segno + non produce alcun effetto sull'operando, nel senso che, come in aritmetica, un valore non preceduto da alcun segno è considerato positivo; ++ e -- incrementano o decrementano di 1 l'operando cui sono applicati. Essi possono precedere **(prefissi)** o seguire **(postfissi)** l'operando:

1. se sono prefissi l'operando viene immediatamente incrementato/decrementato;
2. se sono postfissi l'operando viene prima utilizzato, se compare in un' espressione, e successivamente incrementato/decrementato.

Esempio :

```
int x=100,y,z;
main()
{
        y=++x;              // equivale a  x=x+1;  y=x;    incremento prefisso
        z=x++;              // equivale a  z=x;   x=x+1;   incremento postfisso
        printf ("%d  %d  %d",x,y,z);          // stampa :102,101,101
}
```

Nel caso in cui l'incremento/decremento compare da solo nell'istruzione, ad esempio :

```
i++;    o       ++i;
```

l'espressione prefissa o postfissa sono perfettamente equivalenti.

2.5 Operatori su bit

Operatore	Dichiarazioni	I Operando	II Operando	Operazione	Risultato
Complemento **~**	unsigned char a=85; unsigned char b;	01010101 (a=85)		b= ~x;	10101010 (b=170)
AND **&**	unsigned char a=165; unsigned char b;	10100101 (a=165)	00001111 (15)	b = a & 15;	00000101 (b=5)
OR **|**	unsigned char a=165; unsigned char b;	10100101 (a=165)	00001111 (15)	b = a | 15;	10101111 (b=175)
XOR ^	unsigned char a=132; unsigned char b;	10000100 (a=132)	10000000 (128)	b = a ^ 128;	00000100 (b=4)
SHIFT a sinistra << di n bit (n = II operando) (*)	unsigned char a=132; unsigned char b;	10000100 (a=132)	1	b = a << 1;	00001000 (b=8)
SHIFT a destra >> di n bit (n = II operando) (*)	unsigned char a=132; unsigned char b;	10000100 (a=132)	1	b = a >> 1;	01000010 (b=66)

(*) Nello SHIFT a sinistra (<<) i bit che escono da sinistra sono persi mentre quelli che entrano da destra sono sempre a 0; nello SHIFT a destra (>>) i bit che escono da destra sono persi mentre quelli che entrano da sinistra sono sempre a zero se il dato è di tipo unsigned, se il dato è un valore con segno in alcuni compilatori viene esteso il bit del segno (entrano bit di valore uguale al segno) in altri entra sempre 0.

2.6 Assegnamento

- assegnamento semplice =
  Esempio : a = 4;
- assegnamento composto ***= /= %= += -= <<= >>= &= |= ^=**
  Esempio : a += 4; // equivale a : a = a+4;
- assegnamenti multipli
  Esempio : a = b = c = 0; // equivale a : c=0; b=0; a=0
  a = (b = +=2) * (c=3) // equivale a : b = b+2; c=3; a=b*c;

2.7 Cast : Conversione di tipo esplicita

È possibile forzare esplicitamente il tipo di una variabile tramite l'operatore **cast**:

cast (*nuovo_tipo***)** *nome_variabile***;**

Esempio :

```
.............
int a = 10,b = 3;
float c;
.............
c = a / b;              // il risultato sarà 3.0
c = (float) a / b;      // il risultato sarà 3.3; mediante il cast si
                        // è forzata a (intera) a float : conversione
                        // esplicita; b ( intera) è forzata a float
                        // per conversione implicita.
```

2.8 Operatore sizeof

L'operatore sizeof restituisce la dimensione in byte di una variabile o di un tipo di variabile :

sizeof (*tipo* **);** // restituisce la dimensione in byte
// di *tipo*

sizeof *nome_variabile***;** // restituisce la dimensione in byte di
// *nome_variabile*

2.9 Ordine di valutazione degli operandi di un'espressione

Una situazione come la seguente è ambigua :

```
...........
int x,a=0;
.............
.............
x= (++a + 5)*(a + 1);
........
```

perché se il compilatore valuta per primo l'operando di sinistra si ha il risultato : x= (1 + 5)*(1+1)
se invece il compilatore valuta per primo l'operando di destra si ha: x=(1+5)*(0+1).
Pertanto situazioni del genere devono essere evitate.

2.10 Costanti

Una costante è un qualsiasi numero, carattere, stringa di caratteri che può essere utilizzato come valore in un programma. Una costante non può essere modificata.

1. **Costanti intere**

- **costante decimale :** numero intero decimale la cui prima cifra è diversa da zero;
- **costante ottale :** numero intero ottale la cui prima cifra è obbligatoriamente 0;
- **costante esadecimale:** numero intero esadecimale preceduto dai caratteri 0x o 0X.

2. **Costanti a virgola mobile :** numero reale decimale espresso nella forma :

[*numero*] [**.***numero*] [**E**|**e** [**+**|**-**] *numero*]

numero prima del punto decimale o quello dopo il punto può essere omesso, ma non entrambi; se si utilizza l'esponente il punto può essere omesso.

3. **Caratteri costanti :** carattere racchiuso tra due apici (‘); sono considerati come caratteri costanti, anche se in realtà sono presenti più caratteri, anche le cosiddette **sequenze di escape** , composte dal carattere **backslash** (\) seguito da un carattere associato ad una specifica azione del terminale o della stampante o da una cifra ottale o esadecimale corrispondente al valore ASCII di un carattere qualunque.

Sequenza di escape	**Azione**	**Valore ASCII (decimale)**
\a	BEEP	7
\b	BACKSPACE	8
\f	FORM FEED (salto pagina)	12
\n	NEW LINE (salto riga)	10
\r	CARRIAGE RETURN	13
\t	Tabulazione Orizzontale (HT)	9
\v	Tabulazione verticale (VT)	11
\”	Virgolette	34
\’	Apice	39
\\	Backslash	92
\ddd	Carattere ASCII in ottale	
\xdd...d	Carattere ASCII in esadecimale	

4. **Stringhe costanti :** sequenza di caratteri racchiusa tra virgolette (“); la sequenza dei caratteri è memorizzata in locazioni contigue di memoria ed è automaticamente chiusa dal carattere **NUL** (**\0**). La stringa non può contenere caratteri non stampabili a meno che non vengano rappresentati come una sequenza di escape (tale sequenza viene considerata come un un solo carattere). Per utilizzare una stringa costante che occupa più di una riga basta inserire alla fine di ogni riga un backslash (\): Due o più stringhe costanti separate da spazi vengono concatenate in un’unica stringa:

```
printf (“ concatenare la prima stringa”
        “ con la seconda”);
```

produce: “ concatenare la prima stringa con la seconda”.

2.11 I qualificatori di tipo : const e volatile

- **const** : dichiara un oggetto come “non modificabile”; la variabile qualificata come **const** non può essere modificata; esempio :

```
const int i = 10;          // ad i viene assegnato il valore 10 e non può
                           // più essere modificata
```

- **volatile** : avvisa il compilatore che il valore di una variabile può essere modificato da eventi indipendenti dal programma (ad esempio dal sistema operativo); perciò il valore di questa variabile deve essere sempre letto dalla memoria e bisogna in tal caso evitare di fare dell’ottimizzazioni.

CAPITOLO 3

Input e output dei dati

3.1 Le funzioni per l'input/output

Il C non dispone di istruzioni per l'input/output dei dati. Per assolvere a queste incombenze si avvale di diverse funzioni presenti nelle librerie che corredano il compilatore C. Le funzioni più utilizzate sono :

- **scanf** per l'input dei dati;
- **printf** per l'output dei dati.

La sintassi è simile per le due funzioni :

scanf (*Formato***,&***var*...**);**
printf (*Formato***,***var*...**);**

var : nome della variabile utilizzata nell'input/output; da notare che nella **scanf** il nome della variabile è preceduta dal simbolo **&** (indirizzo).

Formato : stringa che descrive il formato secondo il quale devono essere acquisiti/stampati i dati; esso viene chiamato **specificatore di formato** ed è racchiuso tra virgolette (**"**); la sintassi è:

%[*]\[*ampiezza*][**h**|**l**|**L**]*tipo* per **scanf**

%[*flags*][*ampiezza*][.*precisione*] [**h**|**l**|**L**]*tipo* per **printf**

* : se presente, il dato viene acquisito ma non viene assegnato ad alcuna variabile;

ampiezza : se presente, indica
- per **printf** il numero minimo di caratteri, compreso l'eventuale punto decimale, che dovrà essere occupato dal dato in uscita;
- per **scanf** il numero massimo di caratteri per il dato in input;

.precisione : se presente, indica il numero di cifre dopo il punto decimale che deve essere stampato o nel caso di numeri interi il numero minimo di cifre occupato dal dato;

flags : carattere che indica il tipo di allineamento del dato (giustificazione);

tipo : nella tabella che segue vengono riportati i tipi più utilizzati

TIPO	SIGNIFICATO
%c	carattere singolo
%d	intero decimale con segno
%u	intero decimale senza segno
%o	intero ottale senza segno
%x, %X	intero esadecimale senza segno
%f	double o float in formato decimale con segno
%e, %E	double o float in formato esponenziale con segno
%s	stringa di caratteri

MODIFICATORE DI TIPO DI FORMATO	SIGNIFICATO
H	prima di **d** : specifica **short int** prima di **u** : specifica **short unsigned int**
L	prima di **d** : specifica **long int** prima di **u** : specifica **long unsigned int** prima di **f** o **e** : specifica **double**
L	prima di **f** o **e** : specifica **long double**

Altre funzioni utilizzate per l'input/output dei dati sono :

- **getchar** : permette di acquisire un carattere dall'unità standard di input (**stdin**), che è normalmente la tastiera; sebbene sia una funzione ANSI il suo comportamento può variare da compilatore a compilatore; in alternativa, nel sistema operativo DOS, sono previste le funzioni :
  getch : acquisizione di un carattere dalla **stdin** senza visualizzazione;
  getche : acquisizione di un carattere dalla **stdin** con **echo** su video;
- **putchar** : scrive un carattere sull'unità di uscita standard (**stdout**), che è normalmente il video);
- **putc** : scrive un carattere sull'unita di uscita specificata : **stdout** per l'uscita standard (video), **stdprn** (standard printer) per la stampante.
- **fprintf** : ha la stessa sintassi di printf con in più la possibilità di indirizzare l'output

fprintf (*Output***,** *Formato***,** *var*....**);**

dove *Output* può valere **stdout** per uscita su video o **stdprn** per uscita su stampante.

- **gets** : acquisizione di stringhe dallo **stdin**; a differenza della **scanf** che non permette l'input di stringhe contenenti "spazi", con **gets** ciò è possibile; la stringa ha termine quando viene incontrato il " ritorno a capo" (tasto INVIO);
- **puts** : stampa di una stringa sullo **stdout**;
- **fputs** : stampa di una stringa sul dispositivo indicato (**stdout** o **stdprn**).

In C++, includendo la direttiva **#include <iostream>**

- **cin >> variabile**
- **cout << espressione**

stdin, stdout, stdprn (DOS) e **stderr** (standard error) e **stdaux** (DOS : porta seriale) sono **stream** (dispositivi logici : ogni **stream** si comporta allo stesso modo indipendentemente dal dispositivo fisico cui è associato) definiti nel file intestazione **stdio.h.**

CAPITOLO 4

Istruzioni

4.1 Introduzione

Nell'illustrare le istruzioni del C, verrano utilizzati i termini :

- *espressione*: una qualunque espressione, ivi compreso l'assegnamento; l' *espressione* è VERA se il suo valore èdiverso da 0, è FALSA se il suo valore è uguale a 0; in un assegnamento il valore restituito dall' *espressione* è il valore assegnato al termine di sinistra. Nella maggior parte dei casi *espressione* sarà un'espressione condizionale cioè un confronto tra 2 termini realizzato mediante i soliti operatori di confronto (> maggiore , >= maggiore o uguale, < minore, <= minore o uguale, == uguale, **!=** diverso). Più espressioni condizionali possono essere collegate tra di loro tramite gli operatori logici (**&&** and, || or, **!** not (agisce su un solo operando)).
- *sequenza...* : serie di istruzioni racchiuse tra parentesi graffe (**istruzione composta** : equivale dal punto di vista sintattico ad una sola istruzione); nel caso *sequenza...* è formata da una sola istruzione le parentesi graffe non sono necessarie.

4.2 Selezione

La selezione in C è realizzata tramite l'istruzione

if (*espressione* **)**
    sequenzaA
else
    sequenzaB

se (**if**) *espressione* è VERA verrà eseguita *sequenzaA* altrimenti (**else**), se esiste, verrà eseguita *sequenzaB*. Esempi :

```
/* stampa del valore più grande e del valore più piccolo tra due valori dati in input */
        # include <stdio.h>
        # include <conio.h>
        main ()
          { int a,b,minab,maxab;
            printf ("introdurre il primo valore : ");
            scanf ("%d", &a);
            printf ("\nintrodurre il secondo valore : ");
            scanf ("%d", &b);
            if (a > b)
                { minab=b; maxab=a; }
              else
                { minab=a; maxab=b; }
             printf ( "\ni valori sono nell'ordine  %d  e  %d", minab, maxab);
             getch();
          }
```

```
/* Valore assoluto di un numero dato in input */
# include <stdio.h>
# include <conio.h>
main()
  { int a,b
    printf (" introduci un numero : ";
    scanf ("%d", &a);
    b=a;
    if (b < 0)  b=-b;
    printf (" \nil valore assoluto di %d  è  %d", a,b);
    getch();
  }
```

4.3 Selezione multipla

Viene realizzata mediante l'istruzione di controllo **switch.**

switch (*espressione* **)**
{
case *costante1* **:** *sequenza1*
case *costante2* **:** *sequenza2*
.....
case *costanten* **:** *sequenzan*
default : *sequenza_default*
}

trasferisce il controllo del programma alla *sequenza...* il cui **case** *costante...* coincide con il valore di *espressione*(di tipo int*)*; vengono eseguite in sequenza tutte le altre istruzioni, senza più controllo sul **case**, a meno che non sia presente un'istruzione che trasferisce altrove il controllo (**break** o **goto**).

Se nessuno dei **case** *costante...* soddisfa l'uguaglianza con *espressione* il controllo viene trasferito a **default :** *sequenza_default* , se esiste, o all'istruzione immediatamente successiva allo **switch**.

In effetti molto spesso si vuole esegire uno solo dei **case** , pertanto ogni sequenza viene chiusa dall'istruzione **break** che permette, una volta eseguita la sequenza voluta, di trasferire il controllo all'istruzione immediatamente successiva allo **switch**. Esempio :

```
/* stampa del giorno della settimana in corrispondenza di un valore dato in input */
        # include <stdio.h>
        # include <conio.h>
        main ()
          { int giorno;
            printf (" introdurre un valore : ");
            scanf ("%d",&giorno);
             switch  (giorno )
                 {
                   case 1 : printf ("lunedi");
                            break ;

                    case 2 : printf ("martedi");
```

```
                    break ;
            case 3 : printf (“mercoledi“);
                    break ;
            case 4 : printf (“giovedi“);
                    break ;
            case 5 : printf (“venerdi“);
                    break ;
            case 6 : printf (“sabato“);
                    break ;
            case 7 : printf (“domenica“);
                    break ;
           default : printf (“valore non corretto”);
                    break;
        }
 getch();
}
```

Se nei **case** fosse stato omesso il **break**, per giorno = 5, ad esempio, il risultato sarebbe stato

venerdisabatodomenicavalorenoncorretto

4.4 Iterazione

L’iterazione può essere realizzata nei seguenti modi

- **while (***espressione***)**
  sequenza

  l’esecuzione delle istruzioni indicate in *sequenza* viene ripetuta finchè *espressione* è VERA.
  Esempio :

```
/* calcolo della lunghezza di una parola data in input */
# include <stdio.h>
# include <conio.h>
main()
  {
    int numpar = 0,lett;
     lett = getchar();
     while ( lett != 32)
         {
           numpar++;
           lett = getchar();
         }
     printf (“ la lungezza della parola è %d “, numpar);
     getch();
   }
```

- **do**
  sequenza
  while (*espressione***);**

l'esecuzione delle istruzioni indicate in *sequenza* viene ripetuta finchè *espressione* è VERA.
Esempio :

```
/* calcolo della lunghezza di una parola data in input */
# include <stdio.h>
# include <conio.h>
main()
  {  int numpar = 0,lett;
     do
        { lett = getchar();
          numpar++;
        } while ( lett != 32)
      printf (" la lungezza della parola è %d ", numpar);
      getch();
     }
```

- **for (** *espressione1* **;** *espressione2* **;** *espressione3***)**
  sequenza

espressione1: se presente, inizializza la variabile di controllo del ciclo;
espressione2: se presente, determina quando deve essere eseguita *sequenza;*sono possibili tre casi:

1. se *espressione2* è VERA (diversa da 0) viene eseguita *sequenza*; quindi, se presente, viene valutata *espressione3* che indica come deve essere modificata la variabile del ciclo ogni volta che viene ripetuto;
2. se *espressione2* è omessa viene considerata VERA e l' esecuzione procede come nel caso 1; il ciclo avrà termine quando al suo interno si incontra un 'istruzione che permette di trasferire il controllo del programma al di fuori del ciclo (**break** , **return** , **goto**);
3. se *espressione2* è FALSA (uguale a 0) l'esecuzione del ciclo ha termine e il controllo del programma passa all'istruzione successiva al **for**.

Esempio :

```
/* dare in input 10 valori e stampare, per ognuno di essi il quadrato */
# include <stdio.h>
# include <conio.h>
main()
 {
  int i, a;
  for (i = 0; i < 10; i++)
        {
          printf ("\nintrodurre un valore : ")
          scanf ("%d", &a);
          printf ("     il quadrato è : %d", a*a);
        }
   getch();
  }
```

4.5 Istruzione break

break;

permette di uscire dallo **switch** o dai cicli **while**, **do while**, **for**.

4.6 Istruzione continue

continue;

permette, all'interno dei cicli **while**, **do while**, **for** , di passare all' iterazione successiva saltando tutte le istruzioni del ciclo che la seguono.

4.7 Istruzione nulla

;

può comparire in qualsiasi punto del programma; non viene eseguita alcuna azione da parte del calcolatore.

4.8 Istruzione goto

goto *label;*
.....
.....
label : istruzione;

label è l'etichetta dell' istruzione cui viene trasferito il controllo del programma dall'istruzione **goto**.

4.9 Operatore ?:

risultato = operando1 **?** *operando2* **:** *operando3***;**

questa espressione può essere interpretata come :

* se *operando1* è VERO *risultato = operando2*
* se *operando1* è FALSO *risultato = operando3*

Esempio : trovare il minimo tra i 2 valori a,b

```
.....                                  // soluzione senza l'utilizzo di ?:
int a,b,minimo;
scanf ("%d",&a);
scanf("%d",&b);
if (a>b) minimo=b;
  else minimo=a;
.....
```

```
.....                    // soluzione con l'utilizzo di ?:
int a,b,minimo;
scanf ("%d",&a);
scanf("%d",&b);
minimo = a>b ? b : a;    // se a>b (operando1) è VERA a minimo viene
                         // assegnato b (operando2)
                         // altrimenti viene assegnato a (operando3)
```

4.10 Operatore ,

Consente di valutare 2 o più espressioni in quei contesti in cui ne è permessa una sola. Il risultato dell'espressione è pari al valore dell'operando di destra (ultimo operando). Ad esempio nell'istruzione **for** sono ammesse solo 3 espressioni; grazie all'operatore **,** se ne possono aggiungere altre :

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

l'ultima espressione permette sia di incrementare il contatore (**i++**) sia di assegnare un valore a k (**k=i+10**); il risultato di questa espressione è pari al valore assegnato a k.

CAPITOLO 5

Dati strutturati

5.1 Strutture

- **vettore (o array)** : insieme di dati dello stesso tipo (**elementi** del vettore) ognuno dei quali è individuato da un valore detto **indice** (*).
- **vettore di caratteri (o stringa)** : è un array i cui elementi sono caratteri; la stringa deve essere chiusa (ultimo carattere della stringa) dal carattere **NUL** ('**\0**') (*).
- **vettore multidimensionale (matrice)** : insieme di dati dello stesso tipo ognuno dei quali è individuato da 2 o più valori detti **indici** (*).

(*) In C il primo elemento del vettore ha indice 0, pertanto se n sono gli elementi del vettore, l'ultimo elemento ha indice n-1.

- **record (struct** e **union)** : insieme di dati di tipo diverso ognuno dei quali è individuato da un nome (**campi**). La dichiarazione di **struct** e **union** può essere fatta in uno dei seguenti modi :

struct [*tag*] {*lista_campi*} [*identificatore* [,*identificatore*] ....]**;**
struct [*tag*] [*identificatore* [,*identificatore*] ....]**;**

union[*tag*] **{***lista_campi***}** [*identificatore* [,*identificatore*] ....]**;**
union[*tag*] [*identificatore* [,*identificatore*] ....]**;**

lista_campi : elenco dei **campi** (nome e tipo) che formano il record;
tag : (opzionale) nome del record;
identificatore : variabile cui viene assegnato il tipo record appena definito;

La differenza tra **struct** e **union** sta che mentre nella **struct** ognuno dei campi occupa una ben precisa zona di memoria a seconda del suo tipo, nell'**union** tutti i campi fanno riferimento alla stessa locazione di memoria pertanto la **struct** occuperà una zona di memoria pari alla somma delle occupazioni dei suoi singoli campi, mentre alla **union** sarà riservata una zona di memoria pari alla memoria richiesta dal suo campo più grande.

Per accedere al singolo campo di una **struct** o **union** la sintassi è:

*nome_variabile_struct***.***nome_campo*

*nome_variabile_union***.***nome_campo*

Molto utile, nel caso di **struct** e **union**, si rivela la dichiarazione di tipo **typedef** : essa permette di definire nuovi tipi di variabile o di rinominare i tipi fondamentali; la sua sintassi è

typedef *specificatore_tipo identificatore***;**

specificatore_tipo : definizione del tipo;
identificatore : nuovo nome del tipo;

Esempi di definizione di dati strutturati :

1. int a[10]; // vettore di 10 elementi di tipo int

2. int a[] = {12,21,3}; // vettore di 3 elementi di tipo int, inizializzato all'atto della
// dichiarazione : a[0] è inizializzato a 12, a[1] è inizializzato a 21,
// a[2] è inizializzato a 3

3. char x[10]; // vettore di 10 elementi di tipo char (stringa) : vi si può memorizzare
// una stringa di 9 caratteri al massimo, in quanto l'ultimo carattere
// deve contenere il carattere di fine stringa '\0'

4. char saluto[] = "Ciao"; // stringa di 5 caratteri inizializzata all'atto della dichiarazione :
// saluto[0] conterrà 'C'
// saluto[1] conterrà 'ì
// saluto[2] conterrà 'à
// saluto[3] conterrà 'ò
// saluto[4] conterrà '\0'

5. char a[] = "k", x='k'; // "k" (carattere racchiuso tra virgolette) è una stringa pertanto deve
// essere assegnato ad un vettore di almeno 2 elementi :
// a[0] conterrà 'k', a[1] conterrà '\0';
// 'k' (carattere racchiuso tra apici) è un carattere : puo essere
// assegnato ad una variabile di tipo carattere

6. int x[4] [3]; // matrice bidimensionale di 4 righe e 3 colonne di interi

7. int x[][3] = {
{ 15,21,3 },
{ 7, 32,43 },
}; // matrice bidimensionale di interi di 2 righe e 3 colonne inizializzata

8. char elenco[10] [20]; // vettore di 10 stringhe ciascuna delle quali può essere lunga al
// massimo 20 caratteri (compreso il carattere '\0')

9. char elenco[][20] = {"Rossi Mario", "Verdi Gino" };
// vettore di 2 stringhe; per accedere ad una determinata stringa è
// sufficiente specificare il primo indice : elenco[0] permette di accedere
// alla riga 0 che contiene la stringa "Rossi Mario" ed elenco[1] alla
// riga 1 che contiene la stringa "Verdi Gino"

10. struct PERSONA
{ char nome[20];
char telefono[15];
int eta;
} a,b,c; // a,b,c sono variabili del tipo struct PERSONA; ognuna di esse
// conterrà i campi nome (stringa di 20 caratteri), telefono (stringa di
// 15 caratteri) eta (intero); l'occupazione di memoria di ognuna di esse
// sarà di 37 byte (20+15+2); a.nome : campo nome della variabile a;
// a.telefono : campo telefono della variabile a; a.eta : campo eta della
// variabile a

11. union PERSONA
{ char nome[20];

```
    char telefono[15];
    int eta;
  } a,b,c;          // a,b,c sono variabili del tipo union PERSONA; ognuna di esse
                    // può contenere i campi (uno alla volta) nome (stringa di 20 caratteri),
                    // telefono (stringa di 15 caratteri) eta (intero); l'occupazione di
                    // memoria di ognuna di esse sarà di 20 byte (memoria richiesta dal
                    // campo più grande)
```

12. struct PERSONA

```
  { char nome[20];
    char telefono[15];
    int eta;
  }
      e
struct  PERSONA a,b,c;   // producono lo stesso effetto della dichiarazione 10
```

13. union PERSONA

```
  { char nome[20];
    char telefono[15];
    int eta;
  }
      e
union  PERSONA a,b,c; // producono lo stesso effetto della dichiarazione 11
```

14. typedef struct

```
  { char nome[20];
    char telefono[15];
    int eta;
  } PERSONA ;
      e
PERSONA a,b,c;          // producono lo stesso effetto della dichiarazione 10
```

15. typedef union

```
  { char nome[20];
    char telefono[15];
    int eta;
  } PERSONA;
      e
PERSONA a,b,c;          // producono lo stesso effetto della dichiarazione 11
```

CAPITOLO 6

Puntatori

6.1 Definizione

Il puntatore è una variabile il cui contenuto è l'indirizzo di un oggetto (variabile, funzione); l' operatore **&** permette di ottenere l'indirizzo di un oggetto, mentre l'operatore unario ***** di indirezione permette ad un puntatore di accedere al valore contenuto nella locazione di memoria il cui indirizzo è contenuto nel puntatore stesso (locazione di memoria "puntata" dal puntatore).

Definizione di variabili puntatore :

tipo *******identificatore***;**

Esempi :

1. int a,*p; // dichiarazione di una variabile di tipo intero (a) e di un puntatore ad // interi (p) p cioè può contenere indirizzi di variabili intere

2. char *q; // q è dichiarata come puntatore a caratteri (può contenere indirizzi di // variabili di tipo carattere)

3. float *n; // n è dichiarata come puntatore a float (può contenere indirizzi di // variabili di tipo float)

4.
```
# include <stdio.h>
main()
  { int a,b, *x,*y;          // a e b variabili intere; x e y puntatori ad interi
    scanf ("%d",&a);
    scanf ("%d),&b);
    x = &a;                  // x conterrà l'indirizzo della variabile a
    y = &b;                  // y conterrà l'indirizzo della variabile b
    printf ("%d",*x);        // è equivalente a printf ("%d",a);
    printf ("%d",*y);        // è equivalente a printf ("%d",b);
   }
```

5.
```
int a,*q;                    // a variabile intera; q puntatore ad interi
&a = 10;                     // è errata perché non si può modificare l'indirizzo di un oggetto
*q = 10;                     // è corretta, perché viene assegnato un valore nella locazione di
                             // memoria puntata da q ma i risultati sono imprevedibili perché la
                                    // cella di memoria indirizzata da q potrebbe essere qualsiasi
```

6. const int *i; // puntatore ad un valore costante di tipo int; il puntatore può essere // modificato perché punti ad un altro valore, ma il valore a cui punta // non può essere modificato

7. int *const i; // puntatore costante ad un valore di tipo int; il valore puntato può // essere modificato il puntatore no

6.2 Vettori e puntatori

C'è una stretta relazione tra vettori e puntatori; infatti in C il nome del vettore coincide con l'indirizzo del primo elemento del vettore stesso cioè, fatte le dichiarazioni :

```
int vett[10],*p;
```

si ha

```
vett ≡ &vett[0]
```

pertanto è lecito scrivere :

```
p = vett ;        // è equivalente a p = &vett[0];
```

e si ha

```
*p ≡ vett[0]
*(p+i) ≡ vett[i]
p[i] ≡ *(vett+i)
```

da notare che *(p+i) è diverso da *p+i (in questo caso viene sommato i al contenuto della cella puntata da p, infatti l'operatore di indirezione ha precedenza più alta dell'addizione).

Nella manipolazione di puntatori e nomi di vettori bisogna tener presente che mentre i puntatori sono delle variabili i nomi di vettore sono delle costanti, perciò , se p è un puntatore e vett il nome di un vettore, mentre è lecito scrivere

```
p = vett;  o p++;
```

è errato scrivere :

```
vett = p; o vett++;
```

perché si sta cercando di modificare l'indirizzo di una variabile.

L'uso dei puntatori permette di scrivere programmi molto concisi, qualche volta però a scapito della comprensione; il seguente segmento di programma permette ad esempio di inizializzare a 0 gli elementi di un vettore :

```
.................
int dati[10],*p;
for (p = dati; p < &dati[10]; *p++ = 0);     // p punta inizialmente al primo elemento del
                                             // vettore  (p=dati);
                                             // il ciclo ha termine quando p = &dati[10]
                                             // nell'elemento puntato da p (*p) viene messo il
                                             // valore 0, quindi p viene incrementato
                                             //(incremento postfisso) per puntare
                                                    // all'elemento successivo
```

Le funzioni per la manipolazione di stringhe (vettori di char) fanno largo uso dei puntatori; le più utilizzate di queste funzioni sono:

- **strcpy(***stringa1***,** *stringa2***)** : copia *stringa2* in *stringa1*.
- **strcmp(***stringa1***,***stringa2***)**: confronta *stringa1* con *stringa2*; restituisce **0** se *stringa1* = *stringa2*; un valore **< 0** se *stringa1* < *stringa2*; un valore **> 0** se *stringa1* > *stringa2*.
- **strcat(***stringa1***,***stringa2***)** : concatena *stringa2* a *stringa1*.
- **strlen(***stringa***)** : restituisce il numero di caratteri di *stringa* ('\0' è escluso dal conteggio).
- **atoi(***stringa***)**, **atol(***stringa***)**, **atof(***stringa***)** : convertono *stringa* rispettivamente in un **int**, un **long**, un **double**.

È possibile anche definire un puntatore ad una struttura (struct o union), esempio :

```
typedef  struct
        {  char nome[20];
           int eta;
        } DATI;
DATI a, *p;                 // a è una variabile del tipo DATI e p un puntatore ad una variabile di
                            // tipo DATI
```

per accedere ad un campo di una struttura attraverso un suo puntatore, piuttosto che attraverso il nome di una variabile, basta sostituire all' operatore **.** l'operatore **->**, cosicchè

a.nome ≡ p->nome

Un uso più complesso dei puntatori si ha nei seguenti casi :

```
1. int (*p)[10];          // p è un puntatore ad un vettore di 10 interi
2. int *p[10];            // p è un vettore di 10 puntatori ad interi
3. char **p;              // p è un puntatore ad un puntatore a char
4. int (*p) (int);        // p è un puntatore ad una funzione che restituisce un intero e che ha
                          // come parametro di input un intero
```

CAPITOLO 7

Funzioni

7.1 Definizioni

Una **funzione** può essere definita come un insieme di dati e istruzioni cui è associato un nome.

L' uso delle funzioni offre diversi vantaggi :

- consente di applicare concretamente la metodologia TOP-DOWN;
- consente di scrivere una volta sola le istruzioni che realizzano operazioni che ricorrono frequentemente.

7.2 Le funzioni in C

Ogni programma C è costituito normalmente da diverse funzioni; in ogni caso esso deve contenere almeno la funzione **main**.

La **definizione della funzione** comprende **l'intestazione** della funzione ed il **corpo**; quando si vogliono far eseguire le istruzioni definite nella funzione viene fatta una **chiamata della funzione** specificando il nome della funzione e fornendo ad essa, eventualmente, i dati di cui necessita (**parametri attuali**). La funzione può restituire al programma chiamante un valore (**valore di ritorno**) e può apportare delle modifiche ai dati che gli sono stati passati (forniti alla funzione al momento della sua chiamata). I "contenitori" dei parametri attuali, le variabili cioè destinate, all'interno della funzione, a contenere i dati passati al momento della chiamata della funzione, sono chiamati **parametri formali** e vengono definiti nell'intestazione della funzione; tra **i parametri formali** ed i **parametri attuali** ci deve essere un'esatta corrispondenza, come numero e tipo. L'associazione (**passaggio**) tra parametri fomali e parametri attuali può essere fatta, in C, in due modi :

1. **passaggio per valore** : il **parametro formale** è una **copia** del **parametro attuale**; ogni modifica fatta, all'interno della funzione al parametro formale non si rpercuote sul parametro attuale, perciò, all'uscita della funzione, il parametro attuale avrà lo stesso valore che aveva al momento della chiamata della funzione.
2. **passaggio per indirizzo** : il **parametro formale** assume lo **stesso indirizzo** del **parametro attuale** : tutte e due le variabili (attuale e formale) fanno riferimento alla stessa locazione di memoria, perciò ogni modifica fatta al parametro formale si ripercuote sul parametro attuale.

7.3 Definizione di funzione

[*tipo_valore_ritorno*] *nome_funzione* **(** [*tipo parametro1***,** *tipo parametro2***,**...] **)** ← intestazione

```
{
   .....
   .....                                  ← corpo
    .....
   return [ (valore_ritorno) ];
}
```

se la funzione non restituisce alcun valore *tipo_valore_ritorno* deve essere **void**

analogamente se la funzione non riceve parametri in ingresso cioè non ci sono parametri formali deve essere indicato il parametro **void.**

7.4 Chiamata di funzione

[*variabile* =] *nome_funzione* **(** [*parametro_attuale1***,** *parametro_attuale2*...] **);**

nel caso la funzione restituisce un valore, questo valore può essere assegnato ad una variabile, ovviamente del *tipo_valore_ritorno*; quando la funzione non accetta parametri in ingresso, cioè non ci sono parametri attuali, il nome della funzione deve essere seguito dalla coppia **( )**.

7.5 Dichiarazione di funzione (prototipo)

Se la dichiarazione della funzione precede la sua chiamata il compilatore può controllare che vi sia corrispondenza, in numero e tipo, tra parametri formali e parametri attuali; purtroppo non sempre è possibile far precedere la definizione di funzione alla sua chiamata, è stato perciò introdotto il **prototipo di funzione** che rispecchia l'intestazione della definizione di funzione, tralasciando eventualmente il nome dei parametri formali (*parametro1*,*parametro2*....), ed è chiusa dal **;** :

[*tipo_valore_ritorno*] *nome_funzione* **(** [*tipo* [*parametro1*]**,** *tipo* [*parametro2*]**,**...] **);**

il **prototipo** deve comparire nel programma sorgente prima della chiamata della funzione.

Anche le funzioni della libreria standard (ad esempio **printf** , **scanf**,..) devono avere il loro prototipo; per evitare di doverli riscrivere ogni volta, i prototipi di queste funzioni sono raggruppati in appositi file di testo (**header file** o file intestazione (normalmente hanno l'estensione **.h**) perché compaiono in testa ai programmi) che vengono ricopiati nei programmi che utilizzano le funzioni tramite la direttiva al preprocessore **#include**.

7.6 La funzione main

Anche il **main**, come le altre funzioni, può accettare dei valori e restituire un valore; ma essendo esso la funzione che viene eseguita per prima non riceve/restituisce questi valori da/a un'altra funzione bensi dal/al sistema operativo.

I dati passati al **main** sono gli argomenti presenti nella riga di comando che richiama il programma; questi argomenti sono considerati delle stringhe costanti e sono passati al **main** come *vettore di puntatori* a **char** : ogni elemento del vettore punta ad un argomento nell'ordine in cui sono elencati; il primo corrisponde al nome del programma.

Come **parametri di ingresso** il **main** utilizza :

- **argc** : variabile int che indica il numero degli argomenti;
- **argv** : vettore di puntatori a char che contiene gli indirizzi degli argomenti.

Esempio :

```
/* nome programma demomain.c   */
#include <stdio.h>
void main ( int argc, char *argv[])
{
  int cont;
  printf ("numero argomenti %d\n\n", argc);
  for (cont=0; cont < argc; cont++)
        printf ("argv[%d] = %s\n",cont,argv[cont]);
        }
```

una volta compilato e linkato, dato il comando :

```
demomain uno due tre
```

si avrà l' uscita :

```
numero argomenti  4

argv[0] = demomain
argv[1] = uno
argv[2] = due
argv[3] = tre
```

Il **valore di ritorno**, se esiste, deve essere di tipo **int**. Se non si ha alcun valore di ritorno allora alcuni compilatori richiedono che alla parola **main** sia premessa l'indicazione di tipo **void**. L'eventuale **valore di ritorno** del **main** è un dato che può essere sfruttato dal sistema operativo.

7.7 Variabili globali e variabili locali

Quando si utilizzano le funzioni è bene avere presente la distizione tra **variabili globali** e **variabili locali**; sono **globali** le variabili dichiarate al di fuori di ogni blocco (**livello esterno**), main compreso, sono **locali** le variabili dichiarate all'interno di un blocco (**livello interno**).

Per **blocco** si intende un insieme di dichiarazioni, definizioni ed istruzioni racchiuso tra parentesi graffe. Ci sono due tipi di blocco : **l'istruzione composta** e **la definizione di funzione** (intestazione + corpo della funzione). Un blocco può comprendere altri blocchi, in tal caso il blocco interno si dice annidato; nessun blocco può però comprendere una definizione di funzione.

Le **variabili globali** sono utilizzabili ("**visibili**") in tutto il file sorgente in cui sono dichiarate e conservano il loro valore ("**durata**" o "**periodo di vita**") per tutta la durata dell'esecuzione del programma, mentre le **variabili locali** hanno "**visibilità**" e "**durata**" limitate al blocco in cui sono dichiarate.

Si riportano le definizioni di:

durata (**periodo di vita**) : intervallo di tempo, durante l'esecuzione del programma, nel quale una variabile esiste.

visibilità : porzione di programma nel quale un item (tutto ciò che può essere refenziato per nome : variabili, funzioni, costanti di enumerazione, nomi di tipi, etichette e membri di strutture e unioni, etichette di istruzioni) può essere referenziato per nome.

7.8 Dichiarazione/definizione di variabili e funzioni

Il comportamento delle variabili, per quanto riguarda la durata e la visibilità, puo essere modificato utilizzando nella loro dichiarazione/definizione i cosiddetti **specificatori di classe di memorizzazione** (**storage-class**).

La sintassi della dichiarazione/definizione di una variabile è :

[*classe_memorizzazione*] [*tipo*] *idendificatore* [= *valore_iniziale*][**,**.....]**;**

Gli specificatori sono **extern**, **static**, **auto**, **register**.
Extern e **static** possono venir impiegati anche per modificare la visibilità delle funzioni (il periodo di vita di una funzione è comunque globale).

C'è differenza tra dichiarazione e definizione :

	Significato	**Funzione**
Dichiarazione	Associazione tra nome e attributi di una variabile, funzione o tipo	prototipo
Definizione	Come la dichiarazione ma in più alloca lo spazio in memoria; tutte le definizioni sono dichiarazioni ma non viceversa	intestazione+ corpo della funzione

Le regole su durata e visibilità sono riepilogate nelle 2 tabelle seguenti :

1. Funzioni (il livello può essere solo esterno)

DICH./DEF.	**STORAGE-CLASS**	**DURATA**	**VISIBILITÀ**
Prototipo/Definizione	static	globale	ristretta al file sorgente in cui è definita
Prototipo	extern (storage-class di default)	globale	totale;

2. Variabili

LIV.	DICH./DEF.	STORAGE-CLASS	DURATA	VISIBILITÀ CARATTERITSICHE
Esterno	Dichiarazione	extern	globale	totale ; - rende visibile una variabile definita nel livello esterno di un altro file sorgente - rende visibile una variable prima della sua definizione nel livello esterno dello stesso file sorgente
	Definizione	static	globale	ristretta al file sorgente in cui è definita; è possibile definire un'altra variabile static in un altro file sorgente dello stesso programma con lo stesso nome senza creare conflitti.
Interno	Dichiarazione	extern	globale	blocco ; rende visibile, all'interno del blocco, una variabile definita nel livello esterno di uno dei file sorgenti che compongono il programma
	Definizione	static	globale	blocco; l'inizializzazione di questa variabile viene fatta una sola volta quando ha inizio l'esecuzione del programma.
	Definizione	auto (storage-class di default)	locale	blocco; queste variabili devono essere inizializzate esplicitamente; esse vengono "perse" quando si esce dal blocco e "ricreate" ogni volta che si entra nel blocco.
	Definizione	register	locale	blocco ; chiede al compilatore di memorizzare la variabile in uno dei registri della CPU, se disponibile; si comportano come le variabili auto.

Un caso che non rientra tra quelli elencati nella precedente tabella si ha quando viene dichiarata una variabile nel livello esterno senza indicare lo storage-class e senza alcun valore di inizializzazione , esempio :

```
int i;
```

si possono avere due situazioni :

- se esiste, nel programma, una definizione della stessa variabile, deve essere sottinteso lo storage-class **extern**;
- se non esiste un' altra definizione della stessa variabile allora la variabile viene inizializzata a 0.

7.9 Inizializzazione

- Una variabile dichiarata/definita nel livello esterno, tranne le variabili con storage-class **extern** che non possono essere inizializzate, può essere inizializzata con un valore costante; di default viene inizializzata a 0.
- Le variabili, a livello interno, con storage-class **auto** o **register**, possono essere inizializzate con valori costanti o variabili; se non sono inizializzate il loro valore iniziale è indefinito.

Esempi :

```
/* calcolo superficie e volume */
# include <stdio.h>
# include <conio.h>
int area (int, int , int);          // prototipo della funzione area
main()
  {
    int supertot;
    supertot = area (2,3,4);        // chiamata della funzione area con i parametri
                                    // attuali 2,3,4; il valore restituito dalla funzione
                                    // viene assegnato alla variabile supertot
     .....
    getch();
  }

int area ( int x, int y, int h)     // definizione della funzione area
  {
    int superficie;
    superficie = x*y;
    printf ( " volume : %d ", superficie * h);
    return (superficie);
  }
/* contare le parole di un testo che comtengono un determinato carattere, ad es. 'n' */
# include < stdio.h >
# include <conio.h>
int esisteparola(void);                 // prototipo della funzione esiste parola
int contienechiave(void)                //prototipo della funzione contienechiave
int chiave = 'n', carattere = '  ';     // variabili globali
main ()
        { int apparizioni = 0;
          while ( esisteparola() == 1)
```

```
                    if ( contienechiave() == 1)  apparizioni++;
                  printf( " n. parole : %d", apparizioni);
                  getch();
                }
        int esisteparola()        // definizione della funzione esisteparola
                {
                  while ( carattere == ' ')
                            carattere = getchar();
                   if  ( carattere == '\n' )
                            return (0);
                      else
                            return(1);
                }

        int contienechiave()   // definizione della funzione contienechave
                 {
                  int test = 0;
                   while ( (carattere != ' ' ) &&  ( carattere != '\n') )
                      {
                        if ( carattere == chiave) test =1;
                        carattere = ' ';
                      }
                   return(test);
                 }

/* scambio di due valori : funzione errata */
#include <stdio.h>
#include <conio.h>
void scambia (int, int);                  // prototipo della funzione scambia
main ()
        {
          int a,b;
          printf ("introdurre un valore : ");
          scanf ("%d", &a);
          printf ("\nintrodurre un valore : ");
          scanf ("%d", &b);
          scambia(a,b);
          printf (" \ni valori scambiati sono : %d   %d ", a,b);
          getch();
        }

void scambia (int x, int y)             // definizione della funzione scambia
        {
          int temp;
          temp = x;
          x = y;
          y = temp;
          return;
        }

/* scambio di due valori : funzione corretta */
```

```
# include <stdio.h>
# include <conio.h>
void scambia (int*, int*);              // prototipo della funzione scambia


main ()
        {
          int a,b;
          printf ("introdurre un valore : ");
          scanf ("%d", &a);
          printf ("\nintrodurre un valore : ");
          scanf ("%d", &b);
          scambia(&a,&b);
          printf (" \ni valori scambiati sono : %d   %d ", a,b);
          getch();
        }

void scambia (int *x, int *y)           // definizione della funzione scambia
        {
         int temp;
         temp = *x;
         *x = *y;
         *y = temp;
          return;
        }
```

CAPITOLO 8

File

8.1 Definizioni

Un insieme di dati memorizzato su memoria di massa e identificato da un nome viene chiamato **file**.

Le considerazioni che seguono sono riferite ai cosiddetti **file binari** (file trattati per byte, a differenza dei **file di testo** che possono essere visti come file trattati per "riga").

Le operazioni che si possono fare su un file sono :

- **apertura del file**

FILE * fopen (char ****nomefile*, char ****modo***);**

fopen : funzione che esegue l'apertura; essa restituisce un valore del tipo puntatore a **FILE** (è una struttura predefinita); questo valore rappresenta il nome del file utilizzato nel programma che potremmo chiamare "nome logico del file";

nomefile : è il nome con cui il file viene riconosciuto dal sistema operativo; "nome fisico" del file.

modo : è la modalità secondo cui viene aperto il file

MODO	SIGNIFICATO
r	apertura in lettura (file già esistente)
w	apertura in creazione/scrittura (nuovo file)
a	apertura in aggiunta a partire dalla fine del file. Se il file non esiste viene creato
r+	apertura in lettura/scrittura (file già esistente)
w+	apertura in creazione e lettura/scrittura (nuovo file)
a+	apertura in aggiunta/lettura a partire dalla fine del file. Se il file non esiste viene creato

Al "*modo*" può essere aggiunto il carattere **b** o **t** che specifica se si tratta di un file di testo o binario. Esempio :

```
.....
FILE  *pfile;
.....
pfile = fopen ( "prova.dat", "wb");  //il file binario "prova.dat" viene aperto
                                     // creazione; il suo "nome logico" sarà
                                     // pfile
```

- **chiusura del file**

fclose (*nome_logico_file* **);**

- **lettura/scrittura**

Se, come normalmente avviene, pensiamo ad un file come ad un insieme di record (in C **struct**), le operazioni di lettura scrittura possono essere scritte come :

fread (**&***variabile_record***, sizeof** *variabile_record***,** *1***,** *nome_logico_file***);** // lettura di un record

fwrite (**&***variabile_record***, sizeof** *variabile_record***,** *1***,** *nome_logico_file***);** // scrittura di un record

- **posizionamento**

Facendo anche in queso caso riferimento ad un file come insieme di record, per il posizionamento sul record numero *num_rec,* si ha :

fseek (*nome_logico_file***,** *num_rec**** sizeof** *variabile_record***,** *origine***);**

origine può assumere i seguenti valori :

SEEK_SET : inizio del file
SEEK_END : fine del file
SEEK_CUR : posizione corrente

- **fine file** (**EOF**)

È possibile controllare la fine del file mediante la funzione

feof (*nome_logico_file***);**

a differenza di altri linguaggi di programmazione la fine file non viene sentita se non dopo una lettura del fine file stesso (lettura al di fuori dei limiti attuali del file).

CAPITOLO 9

Il preprocessore C

9.1 Direttive

Il preprocessore è un programma richiamato durante la prima fase della compilazione; esso ha lo scopo di modificare il testo del programma sorgente secondo delle **direttive** specificate nel file da compilare.

Ogni **direttiva** è preceduta dal simbolo **#** e non deve essere chiusa dal **;**.

Su una riga può essere specificata una sola direttiva.

Le **direttive** riconosciute dal **preprocessore**, secondo l'ANSI C, sono :

- **#include** : **#include <** *path-spec*> o **#include "***path-spec***"** sostituisce la riga che la contiene con il contenuto del file indicato da *path-spec*.
- **#define** : serve per associare un nome ad una costante, un'espressione, un'istruzione; nel caso il nome è associato ad una costante si parla di **costante palese** altrimenti di **macro**. La sua sintassi è

  #define *identificatore* [(*lista_parametri*)] [*testo sostitutivo*]

  Esempi :

```
#define PIGRECO   3.14          // nel programma all'identificatore PIGRECO
                                // verrà sostituito il valore 3.14
#define SE        if            // l'identificatore SE verrà sostituito con  if
#define BEGIN     {             // l'identificatore BEGIN verrà sostituito con {
#define END       }             // l'identificatore END verrà sostituito con }
#define ALLORA                  // in questo caso manca il testo sostitutivo :
                                // l'identificatore ALLORA verrà eliminato dal
                                // programma
#define MIN(x,y)  ( (x) < (y)  ?  (x)  :  (y) )   // esempio di macro : se nel
                                                  // programma si  incontrerà
                                                  // MIN (-3, -8) si avrà
                                                  // l'espansione in
                                                  // ( (-3) < (-8) ? (-3) : (-8) )
```

- **#undef** : **#undef** *identificatore* : fa si che la definizione di *identificatore* non sia più valida.
- **#if**, **#elif**, **#else**, **#endif** : permettono di determinare il comportamento del compilatore o del preprocessore medesimo in base al risultato di un test.
- **#ifdef**, **#ifndef** : son rispettivamente l'unione di **#if** e **defined** e l'unione di **#if** e **not defined**.
- **#pragma** : **#pragma** *istruzione* a differenza delle altre questa direttiva non viene rimossa dal preprocessore e sostituita essa infatti esprime delle istruzioni per il compilatore : l'azione in essa indicata viene eseguita durante la compilazione.
- **#error** : **#error** *messaggio* :viene utilizzata durante la fase di messa a punto del programma; genera un messaggio e il compilatore esamina il programma per verificare la presenza di errori senza generare il file oggetto.
- **#line** : consente di modificare le variabili predefinite del preprocessore _LINE_ e _FILE_ utilizzate, in fase di compilazione, per memorizzare il numero di riga ed il file sorgente cui appartiene.

CAPITOLO 10

I file include del Borland C++

LINGUAGGIO	FILE INTESTAZIONE	DESCRIZIONE
	alloc.h	funzioni di gestione della memoria(allocazione, deallocazione, etc...)
ANSI C	assert.h	macro di debugging assert
C++	bcd.h	classe C++ bcd, operatori sovraccaricati, funzioni matematiche per la classe bcd
	bios.h	chiamata delle routine del BIOS del PC IBM
C++	complex.h	funzioni matematiche per i numeri complessi
	conio.h	funzioni per le routine di I/O della console DOS
ANSI C	ctype.h	funzioni per la gestione dei caratteri
	dir.h	funzioni per la gestione delle directory e dei percorsi
	dos.h	costanti e dichiarazioni per le chiamate al DOS o per operazioni specifiche del microprocessore 8086
ANSI C	errno.h	costanti mnemoniche per i codici di errore
	fcntl.h	costanti simboliche usate in connessione con la libreria di routine open
ANSI C	float.h	parametri per le routine che operano con i numeri in virgola mobile
C++	fstream.h	classi C++ per le routine di inpu/output
C++	generic.h	macro per la dichiarazione delle classi
C++	graphics.h	prototipi delle funzioni grafiche
	io.h	strutture e dichiarazioni per le routine di I/O di basso livello
C++	iomanip.h	manipolatori dei canali di I/O del C++ e macro per la creazione dei manipolatori parametrizzati
C++	iostream.h	routine I/O fondamentali del C++ (ver. 2.0)
ANSI C	limits.h	parametri d'ambiente, limitazioni di compilazione e valori utilizzabili per i numeri interi
ANSI C	locale.h	informazioni specifiche per la nazione e la lingua
ANSI C	math.h	funzioni matematiche
	mem.h	funzioni di manipolazione della memoria
	process.h	funzioni per la gestione dei processi
ANSI C	setjmp.h	funzioni longjmp e setjmp
	share.h	funzioni per la condivisione dei file

LINGUAGGIO	FILE INTESTAZIONE	DESCRIZIONE
ANSI C	signal.h	funzioni signal e raise
ANSI C	stdarg.h	gestione numero variabile di argomenti di funzioni

ANSI C	stddef.h	serie di tipi di macro molto comuni
ANSI C	stdio.h	gestione I/O
C++	stdiostr.h	classi di canali C++ utilizzate dai file stdio
ANSI C	stdlib.h	routine di uso molto comune
C++	stream.h	routine di I/O per i canali C++ (ver.1.2)
ANSI C	string.h	manipolazione di stringhe e della memoria
C++	strstrea.h	classi di canali C++ per gli array di byte in memoria
	sys\stat.h	costanti simboliche per l'apertura e la creazione di file
	sys\timeb.h	funzione ftime
	sys\timebs.h	tipo time_t utilizzato dalle funzioni che operano sui valori di tempo
ANSI C	time.h	routine per la gestione dell'ora
	values.h	costanti varie (per compatibilità con UNIX Syst. V)

CAPITOLO 11

Inclusione di routine Assembly in un programma C

11.1 Codice Assembly "in linea"

Se la routine assembly è inserita all'interno di un programma C basta racchiudere il codice assembly tra le parentesi graffe e premettere la parola **asm.**

Esempio :

```
/* cancellazione dello schermo tramite istruzioni assembly  e
   stampa del quadrato di un nemero dato in input              */

# include <stdio.h>
# include <conio.h>
main ()
     {
        int i;
        asm                              // routine assembly
                {
                  mov cx,0               // angolo superiore finestra
                  mov dx, 2479h          // angolo inferiore finestra
                  mov bh, 7              // attributi normali dello schermo
                  mov ax, 0600h          // interrupt bios
                  int 10h                // richiama interrupt
                  mov dx, 0000h          // riporta il cursore in alto a sinistra
                  mov bh, 00h
                  mov ax, 0200h
                  int 10h
                }
        printf("introdurre un valore : ");
        scanf ("%d", &i);                        // input del valore
        printf ("\n il quadrato è : %d", i*i);   // stampa il quadrato
        getch();
   }
```

11.2 Moduli C e Assembly distinti

Quando si utilizzano moduli assembly non "in linea" essi vengono trattati come una funzione esterna e richiamati dal programma principale C. La routine C deve essere, perciò, dotata di un prototipo come una qualsiasi funzione C, inoltre si dovrà indicare l' elenco degli argomenti che vengono passati alla funzione.

Il prototipo della funzione assembly potrebbe, ad esempio, avere il seguente aspetto :

```
extern  int somma(int, int, int);
```

e la funzione assembly , a sua volta, potrebbe avere la seguente intestazione :

```
somma  PROC  C NEAR  n1,n2,n3: WORD
```

dove “C” indica al compilatore che gli argomenti vengono passati nello stack da destra verso sinistra (in alternativa si può utilizzare la modalità PASCAL : gli argomenti vengono passati da sinistra verso destra); la procedura è NEAR se si utilizza il modello di memoria small, per gli altri modelli (medium, large o huge) si deve dichiarare come FAR.

La funzione assembly restituisce automaticamente il contenuto del registro ax.

Esempi :

```
/* cancellazione dello schermo tramite una funzione C esterna
   e stampa del quadrato di un numero dato in input                */
```

- Modulo C

```
# include <stdio.h>
# include <conio.h>
extern void pulisci (void);                    //prototipo funzione assembly
main ()
     {
        int i;
        pulisci();                             //chiamata della funzione
        printf(“introdurre un valore : “);
        scanf (“%d”, &i);                      // input del valore
        printf (“\n il quadrato è : %d”, i*i); // stampa il quadrato
        getch();
    }
```

- Modulo Assembly

```
DOSSEG                  ; usa la struttura dei segmenti Intel
.MODEL small            ; imposta il modello di memoria
.8086                   ; istruzioni  8086

.CODE
PUBLIC  C pulisci
pulisci  PROC  C  NEAR

                mov cx,0            // angolo superiore finestra
                mov dx, 2479h       // angolo inferiore finestra
                mov bh, 7           // attributi normali dello schermo
                mov ax, 0600h       // interrupt bios
                int 10h             // richiama interrupt
                mov dx, 0000h       // riporta il cursore in alto a sinistra
                mov bh, 00h
                mov ax, 0200h
                int 10h
                ret                 // ritorno al programma chiamante
pulisci     ENDP
            END
```

Basterà compilare insieme con il comando **bcc** i moduli C e Assembly per ottenere un unico programma eseguibile.

APPENDICE A

A.1 Interrupt.

Il processore deve essere in grado di gestire diversi eventi che possono verificarsi in tempi non prevedibili.
Molti di questi eventi sono generati dall'hardware; alcuni di essi sono :
- un segnale della tastiera quando un tasto viene premuto;
- un segnale della stampante di fine carta;
- un segnale di fine trasferimento dati su disco;
- un segnale per un chip di memoria difettoso.

Occasionalmente un evento non prevedibile puo' essere generato dal software :
- un programma che cerca di dividere un valore per zero.

La questione e' : come il processore puo' gestire questi eventi senza interferire con il lavoro che sta eseguendo ?
Possiamo comprendere il comportamento del processore facendo riferimento ad una situazione tratta dalla vita di tutti i giorni :
supponiamo di essere in cucina intenti a preparare una ricetta; possiamo essere distratti dal nostro lavoro da tutta una serie di eventi non prevedibili : suonano alla porta; il telefono squilla; etc.. ; in ogni caso dovremo in qualche modo gestire queste distrazioni senza perdere di vista la preparazione della nostra ricetta.
I comportamenti possibili sono due :
1) ignorare gli eventi quando essi si verificano ed interrompere ad intervalli regolari il nostro lavoro per verificare se qualcosa richiede la nostra attenzione;
2) gestire gli eventi nel momento in cui essi si verificano: ma in tal caso bisogna far in modo che riusciamo a riprendere il nostro lavoro esattamente dal punto in cui era stato interrotto.
Il processore segue questa seconda strada : esso e' predisposto a riconoscere un qualsiasi segnale che indica che si e' verificato un evento che richiede la sua immediata attenzione. Tale segnale viene chiamato *interrupt* e le azioni che il processore intraprende per gestire l'evento vengono chiamate *servizio dell'interrupt*. Sebbene cio' possa sembrare un meccanismo troopo complicato, il processore lavora cosi' velocemente che il riconoscimento ed il servizio dell'interrupt avvengono in pochissime frazioni di secondo.

A2. Come viene servito un interrupt.

Una volta che un interrupt e' stato recepito, il processore esegue una speciale procedura chiamata gestore dell'interrupt; una volta che tale procedura e' stata eseguita il processore riprende il programma in esecuzione. Dal punto di vista del processore un gestore di interrupt non e' dissimile da una normale procedura solo che in questo caso bisogna salvare un numero maggiore di informazioni per poter riprendere il programma.
Quando si verifica un interrupt il processore esegue le seguenti azioni :
- finisce l'esecuzione dell'istruzione corrente;
- salva il registro dei flag nello stack;
- salva nello stack l'indirizzo di segmento e di offset della successiva istruzione;
- calcola l'indirizzo della opportuna procedura di gestione dell'interrupt;
- inizia l'esecuzione di questa procedura.

A3. La INTERRUPT DESCRIPTOR TABLE.

Quando si verifica un interrupt il processore deve essere in grado di reperire l'indirizzo di inizio della procedura appropriata.
Il sistema e' stato progettato in modo tale che ogni interrupt e' identificato da un numero chiamato *vettore di interrupt*.
Il DOS memorizza una tabella di indirizzi, chiamata *interrupt descriptor table*; ognuno degli indirizzi memorizzati nella tabella punta ad una diversa procedura di gestione degli interrupt. Questa tabella e' memorizzata a partire dall'indirizzo 0000H fino all'indirizzo 03FFh (1024 byte): siccome ogni indirizzo completo richiede 4 byte nella tabella sono memorizzati 256 indirizzi diversi: pertanto il DOS puo' gestire 256 interrupts diversi numerati da 0 ad FFh : l'indirizzo della procedura di gestione dell'n-esimo interrupt e' memorizzato alla locazione 4 x n.
La interrupt descriptor table e' creata automaticamente nell'atto in cui viene caricato il DOS.
Due importanti osservazioni :
1) in linea di principio gli indirizzi contenuti nella IDT possono essere cambiati : se si scrive una particolare procedura di gestione di un interrupt, si deve sostituire, per quell' interrupt, l'indirizzo presente nella IDT con l'indirizzo della procedura;
2) alcune "entries" nella IDT non contengono indirizzi di procedure di gestione interrupt bensi indirizzi di tabelle di informazioni utilizzate dal sistema operativo.

AVVERTENZA: la descrizione appena fatta si riferisce alla gestione degli interrupt da parte del DOS in "real mode" o "virtual 86 mode". Il DOS in "protected mode" o altri sistemi operativi lavorano in modo differente.

A4. Come possono essere usati gli interrupts dal programmatore.

Gli interrupts hanno due scopi :
- permettere al computer di rispondere il piu' immediatamente possibile agli eventi che accadono in tempi non prevedibili. Nella maggior parte dei casi tali eventi sono causati da segnali che provengono dai vari dispositivi hardware. Cosicche' gli interrupt sono uno strumento che permette al processore di supportare l'hardware.
- possono essere anche visti come un meccanismo che permette al processore di richiamare una procedura conosciuta solo da un particolare numero (il vettore di interrupt): il programma puo' cosi richiamare una procedura senza conoscere il nome o l'indirizzo.

Perche' il programma potrebbe aver bisogno di richiamare tali procedure? La risposta e' che il DOS ed il BIOS offrono diversi servizi al programma. Per trarre vantaggio di tali servizi il programma deve richiamare l'appropriata procedura, ma non si conosce l'indirizzo di tale procedura; la soluzione sta nell'assegnare ad ognuno di tali servizi un numero che corrisponde ad un vettore di interruzione : il programma puo' richiedere il servizio tramite l'appropriato interrupt comunicato con l'istruzione INT

INT *vettore di interrupt*

Un interrupt generato da un dispositivo viene chiamato interrupt hardware o interrupt esterno. Un interrupt generato da un programma in esecuzione viene chiamato interrupt software o interrupt interno.

A5. Il BIOS.

Il BIOS (Basic Input/Output System)e' una serie di programmi memorizzati su ROM che svolge tre importanti funzioni:
1) contiene il POST (Power-On Self Test) che viene eseguito automaticamente all' accensione del computer per il test dei componenti.
2) contiene degli speciali programmi chiamati device drivers che funzionano da interfaccia con i vari dispositivi hardware. Essi fanno in modo che il programmatore sia svincolato dalla conoscenza delle caratteruistiche dei vari dispositivi hardware.Oltre a questi device driver il DOS permette di installare tutta una serie di device drivers tramite il comando DEVICE specificato nel CONFIG.SYS (Es. ANSI.SYS).
3) contiene una serie di servizi richiamabili tramite un interrupt software.

Le routine del BIOS sono :

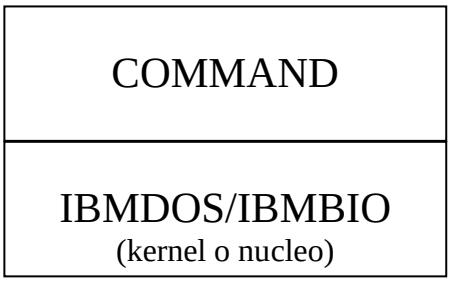
10h servizi del video
11h controllo del dispositivo
12h memoria disponibile
13h servizi per i dischi
14h servizi per le porte seriali
15h I/O su cassetta
16h servizi per la tastiera
17h servizi per la stampante
18h attivazione Basic in ROM
19h avviamento da disco
1Ah servizi per l'ora

A6. Il DOS.

Il BIOS se da una parte ha il vantaggio della velocità dall' altra presenta due grossi inconvenienti :
- e' strettamente legato all'hardware della macchina
- controlla e gestisce in modo primitivo i dispositivi (ad esempio non ha routine per strutturare, gestire e manipolare i dati).

Il DOS puo' essere anche visto come una collezione di servizi richiamabili tramite un interrupt software.

L'architettura del DOS è la seguente :



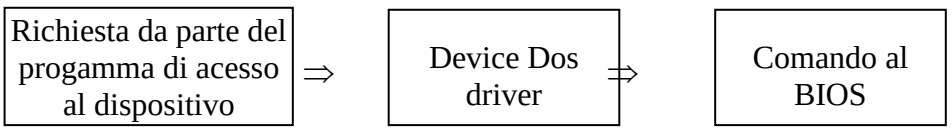
Il file IBMBIO e' la parte del DOS orientata al livello immediatamente inferiore cioe' al Bios mentre il file IBMDOS e' la parte orientata al livello superiore cioe' quello applicativo.
Il COMMAND e' l'interprete dei comandi. E' suddiviso in due parti : la parte transiente in cui vengono caricati di volta in volta i programmi da eseguire e la parte residente in cui i comandi vengono elaborati.

Le routine del DOS sono :

20h fine programmma
21h chiamata di funzione
22h indirizzo terminale
23h indirizzo di uscita CTRL-C
24h vettore per errori fatali
25h lettura settore
26h scrittura settore
27h termina ma rimane residente

A6.1. I device drivers del DOS.

Una istruzione a livello di sistema operativo viene trasformata in una serie di comandi elementari da inviare al BIOS : esiste uno strumento software che permette di fare questa operazione. I programmi predisposti per tale compito prendono il nome di device drivers cioe' piloti di dispositivo.
Un device driver e' un insieme di routine scritte a basso livello e concepite per interfacciare il sistema operativo con il BIOS e quindi con il dispositivo



I *driver standard* del DOS son contenuti nel file IBMBIO (oppure IO.SYS).
Il dos assegna ad ogni driver lo stesso nome del dispositivo che esso gestisce :
COM1 : porta seriale 1
LPT1 : porta parallela 1
CON : console (monitor + tastiera)
A,B,C : dischi a,b,c

Tutti i driver vengono caricati in memoria ed inizializzati all'avviamento del sistema; piu' precisamente quando il BIOS carica il file IBMBIO per trasferirgli il controllo.
I device driver possono essere suddivisi in
- device driver per dispositivi a carattere (gestione delle periferiche con le quali la comunicazione avviene un byte alla volta : console, porta seriale...)
- device driver per dispositivi a blocchi (driver per dispositivi in grado di trattare blocchi di dati : dischi)
- driver senza dispositivo : driver per la gestione dei dispositivi virtuali ad es. disco D o il RAM disk.

I *driver installabili* sono driver personalizzati, inseriti con il comando DEVICE del Config.sys.

I driver sono organizzati nell'area di memoria utilizzata dal DOS in una lista il cui primo elemento e' il driver NUL.

A7. LE FUNZIONI DEL DOS E DEL BIOS.

Per un elenco dettagliato e completo consultare The Ms-Dos Encyclopedia edita da Microsoft Press.

CHIAMATE FUNZIONI BIOS

- *INTERRUZIONE 10h servizi video*

Funzione 00h:impostazione modalità video

Input: AH = 00h
AL = modalità

Risoluzione	Buffer video		
00h	testo a 16 tonalità grigio	40x25	B000:8000h
EGA : 64 colori			
01h	testo a colori 16/8	"	"
EGA:64 colori			
02h	testo a 16 tonalità grigio	80X25	"
EGA:64 colori			
03h	testo a colori 16/8	"	"
EGA:64 coloeri			
04h	grafica a 4 colori	320X200	"
05H	grafica a 4 ton.di grigio	"	"
06H	grafica a 2 ton.di grigio	640x200	"
07H	testo monocromatico	80x25	B000:0000H
08H	grafica a 16 colori	160x200	"
09H	grafica a 16 colori	320x200	"
0AH	grafica a 4 colori	640x200	"
0DH	grafica a 16 colori	320x200	A000:0000H
0EH	grafica a 16 colori	640x200	"
0FH	grafica monocromatica	640x350	"
10H	grafica a 16/64 colori	640x350	"

Output : nessuno

Funzione 01h: impostazione dimensione e forma del cursore

Input AH =01H
CH = inizio scansione riga
CL = fine scansione riga
Nota: CH<CL cursore in blocco unico
CH>CL cursore diviso in due blocchi
CH=20h nessun cursore

Output : nessuno

Funzione 02H: impostazione posizione cursore

Input: AH=02H
BH=pagina video (0 nella grafica)
DH=numero riga
DL=numero colonna

Output : nessuno

Funzione 03h: lettura forma, dimensione e posizione cursore

Input: AH=03H
BH=pagina video

Output: CH=inizio scansione riga
CL=fine scansione riga
DH=numero riga
DL=numero colonna

Funzione 05H : selezione pagina attiva

Input: AH = 05H
AL = numero pagina 00-07H testo 40 colonne
00-03H testo 80 colonne
altri grafica EGA
Nota : ogni pagina=2kb per trsto 40 colonne, 4kb per testo 80 colonne

Output : nessuno

Funzione 06H : scorrimento finestra verso l'alto/il basso

Input: AH = 06H verso l'alto
= 07H verso il basso
AL = numero di linee da far scorrere (00h pulisce il video)
BH = attributi di visualizzazione per linee vuote
CH = n.riga angolo in alto a sinistra
CL = n.colonna angolo in alto a sinistra
DH = n.riga angolo in basso a destra
DL = n.colonna angolo in basso a destra

Output:nessuno

Funzione 08H : lettura carattere ed attributi nella posizione del cursore

Input: AH = 08H
BH = pagina video (solo per pagine testo)

Output: per testo AH = attributi di colore del carattere
AL = caratteri ASCII dalla posizione corrente
per grafica AL = caratteri ASCII (00H se non compatibili)

Funzione 09H : scrittura carattere ed attributi.

Input: AH = 09H
AL = carattere ASCII da visualizzare
BH = pagina video
BL = attributi del carattere (modalità testo) o colore carattere in grafica
CX = n.volte che il carattere deve essere visualizzato
Nota : la posizione del cursore resta invariata

Output: nessuno

Funzione 0AH : scrittura carattere

Input: AH = 0AH
AL = carattere ASCII da visualizzare
BH = pagina video
BL = colore di base del carattere in grafica (non usato in testo)
CX = n.volte che il carattere deve essere visualizzato
Nota : posizione cursore invariata

Output: nessuno

Funzione 0FH : lettura modalità video corrente
Input: AH = 0FH

Output: AH = caratteri per riga (20,40,80)
AL = modalità video corrente
BH = pagina video attiva

- *INTERRUZIONE 11h elenco periferiche*

Output: AX = parola elenco periferiche (PPMURRUFFVVUUCI)

PP	numero stampanti
M	1 se è presente modem interno
RRR	numero porte seriali installate
U	non usato
FF	numero dei floppy drives meno 1
VV	modalità video iniziale
	00 riservato
	01 colore 40x25
	10 colore 80x25
	11 monocromatico 80x25
C	1 se è presente il coprocessore matematico
I	1 se è presente il disco di caricamento

- *INTERRUZIONE 12h memoria disponibile (in Kb)*

Output: AX = memoria disponibile in Kb

- *INTERRUZIONE 14h servizi porte seriali*

Funzione 00h:inizializzazione parametri porte

Input: AH = 00h
AL = parametri porta (BBBPPSCC)
BBB velocità in baud
000 110 baud
001 150 baud
010 300 baud
011 600 baud
100 1200 baud
101 2400 baud
110 4800 baud
111 9600 baud
PP codice parità
00 nessuno
01 dispari
10 nessuno
11 pari
S numero dei bit di stop
0 1 bit di stop
1 2 bit di stop
CC grandezza carattere
00 non usato
01 non usato
10 7 bit
11 8 bit
DX = numero porta seriale (0=COM1)

Output: nessuno

Funzione 01h:trasmissione di un carattere

Input: AH = 01h
AL = carattere da trasmettere
DX = numero porta seriale

Output: AH = codice di errore (vedi funzione 03h ; 00h nessun errore)

Funzione 02h:ricezione di un carattere

Input: AH = 02h
DX = numero porta seriale

Output: AL = carattere da trasmettere
AH = codice di errore (vedi funzione 03h ; 00h nessun errore)

Funzione 03h:lettura stato della porta seriale

Input: AH = 03h
DX = numero porta seriale

Output: AX = stato della porta seriale (errore segnalato dal bit a 1) :
AH : b7 tempo scaduto
b6 registro a scorrimento vuoto
b5 registro di mantenimento vuoto
b4 rilevato break
b3 errore di frame
b2 errore di parità
b1 errore di over run
b0 dati pronti
AL : b7 rilevato segnale sulla linea
b6 indicatore di chiamata
b5 dati disponibili
b4 abilitazione a trasmettere
b3 rilevato segnale di ricezione sulla linea delta
b2 chiamata sul fronte di salita
b1 impostazioni dati disponibili delta
b0 delta pronto per invio

- *INTERRUZIONE 16h servizi tastiera*

Funzione 00h:lettura carattere dal buffer di tastiera

Input: AH = 00h

Output: se è un carattere ASCII
AH = codice scansione tastiera standard
AL = codice ASCII del tasto premuto
se è un carattere ASCII esteso
AH = codice ASCII esteso
AL = 00h

Funzione 01h:controlla se è presente un carattere nel buffer di tastiera

Input: AH = 01h

Output: Flag Z = 1 nessun carattere presente
0 carattre presente
AH e AL = vedi funzione 00h

Funzione 02h:lettura stato tastiera
Input: AH = 02h

Output: AL = stato della tastiera
b0 attivo = shift destro attivo
b1 attivo = shift sinistro attivo
b2 attivo = Ctrl attivo
b3 attivo = Alt attivo
b4 attivo = Scroll Lock attivo
b5 attivo = Num Lock attivo
b6 attivo = Maiuscole attivo
b7 attivo = tasto Insert attivo

- INTERRUZIONE 17h servizi stampante

Funzione 00h:invio carattere alla stampante
Input: AH = 00h
AL = carattere
DX = numero stampante (0=LPT1)

Output: AH = stato della porta (vedi funzione 02h)

Funzione 01h:inizializzazione stampante
Input: AH = 01h
DX = numero stampante (0=LPT1)

Output: AH = stato della porta (vedi funzione 02h)

Funzione 02h:lettura stato stampante
Input: AH = 02h
DX = numero stampante (0=LPT1)

Output: AH = stato della stampante (segnalato dal bit a 1) :
AH : b7 stampante libera
b6 riconoscimento stampante
b5 mancanza carta
b4 stampante selezionata
b3 errore I/O
b2 non usato
b1 non usato
b0 tempo scaduto

- *INTERRUZIONE 19h reset computer* (avvio a caldo)

- *INTERRUZIONE 1Ah lettura/impostazione data/ora*

Funzione 00h:lettura contatore orologio
Input: AH = 00h

Output: AL = segnale di mezzanotte
CX = parola più significativa contatore di tick
DX = parola meno significativa contatore di tick

Funzione 01h:impostazione contatore orologio
Input: AH = 01h
CX = parola più significativa contatore di tick
DX = parola meno significativa contatore di tick

Output: nessuno

Funzione 02h:lettura orologio in tempo reale
Input: AH = 02h

Output: AH = 0 orologio attivo
1 orologio fermo
CH = ora in codice BCD
CL = minuti in codice BCD
DH = secondi in codice BCD

Output: nessuno
Funzione 03h:impostazione orologio in tempo reale
Input: AH = 03h
CH = ora in codice BCD
CL = minuti in codice BCD
DH = secondi in codice BCD
DL = 00h ora solare
01h ora legale

Output: nessuno

Funzione 04h:lettura data da orologio in tempo reale
Input: AH = 04h

Output: Flag C = 0 orologio attivo
1 orologio fermo
CH = secolo in codice BCD
CL = anno in codice BCD
DH = mese in codice BCD
DL = giorno in codice BCD

Funzione 05h:impostazione data orologio in tempo reale

Input: AH = 05h
CH = secolo in codice BCD
CL = anno in codice BCD
DH = mese in codice BCD
DL = giorno in codice BCD

Output: nessuno

Funzione 06h:impostazione allarme

Input: AH = 06h
CH = ore in codice BCD
CL = minuti in codice BCD
DH = secondi in codice BCD

Output: Flag CD = stato :
0 = operazione riuscita
1 = allarme già predisposto o orologio fermo

Funzione 07h:reset allarme

Input: AH = 07h

Output: nessuno

CHIAMATE A FUNZIONI DOS.

- INTERRUZIONE 21h

Funzione 01h:input carattere da tastiera con eco
Input: AH = 01h

Output: AL = carattere letto (codice ASCII)

Funzione 02h:output carattere
Input: AH = 02h
DL = carattere (codice ASCII)

Output: nessuno

Funzione 03h:stampa carattere
Input: AH = 03h
DL = carattere (codice ASCII)

Output: nessuno

Funzione 08h:input senza eco
Input: AH = 07h

Output: AL = carattere (codice ASCII)

Funzione 09h:visualizza una stringa
Input: AH = 09h
DS:DX = segment:offset della stringa da leggere

Output: nessuno

Funzione 0Ah:input di una stringa in un buffere in memoria
Input: AH = 0Ah
DS:DX = segment:offset del buffer di memoria in cui memorizzare la stringa letta

Output: nessuno

Funzione 25h:imposta vettore di interruzione
Input: AH = 25h
AL = numero dell'interruzione
DS:DX = segment:offset della routine di gestione interruzione

Output: nessuno

Funzione 31h:termina il processo e rimane residente
Input: AH = 31h
AL = codice di ritorno
DX = numero di paragrafi di memoria da riservare per il processo residente

Output: nessuno

Funzione 35h:legge vettore di interruzione
Input: AH = 35h
AL = numero dell'interruzione

Output: ES:BX = segment:offset della routine di gestione interruzione specificata in AL

Funzione 4Ch:termina il processo con codice di ritorno
Input: AH = 4Ch
AL = codice di ritorno

Output: nessuno