

INTRODUZIONE ALLA PROGRAMMAZIONE IN C

by Fiorella Sartori

CISCA - Presidio Informatico Telematico Multimediale
della Facoltà di Scienze MM.FF.NN.
Università degli Studi di Trento

Indirizzare eventuali consigli o suggerimenti all'indirizzo e-mail
fiorella@science.unitn.it

La versione ipertestuale di questo documento è presente all'indirizzo
<http://alpha.science.unitn.it/~fiorella/guidac/indexc.html>

Ultimo aggiornamento: 17.11.97

INDICE

-
01. Caratteristiche del linguaggio C
 02. Storia del linguaggio C
 03. Primo approccio
 04. Compilazione di un programma C
 05. Struttura di un programma C
 06. Variabili
 - 06.01. Stampa ed input di variabili
 07. Operatori
 - 07.01. Operatori aritmetici
 - 07.02. Operatori di confronto
 - 07.03. Operatori logici
 - 07.04. Operatori di basso livello
 - 07.04.01. Operatori di bitwise
 - 07.04.02. Bit Fields
 - 07.05. Ordine di precedenza degli operatori
 08. Strutture di controllo
 - 08.01. If
 - 08.02. Operatore "?"
 - 08.03. Switch
 - 08.04. For
 - 08.05. While
 - 08.06. Do-While
 - 08.07. Break e Continue
 09. Arrays
 - 09.01. Array singoli e multidimensionali
 - 09.02. Stringhe
 10. Funzioni
 - 10.01. Funzioni "void"
 - 10.02. Funzioni ed array
 - 10.03. Prototipi di funzioni
 11. Ulteriori tipi di dati
 - 11.01. Strutture
 - 11.02. Unioni
 - 11.03. Type-casting
 - 11.04. Enumerated Types
 - 11.05. Variabili statiche
 12. Errori comuni in C
 - 12.01. Assegnazione (=) al posto di confronto (==)
 - 12.02. Passaggio dell'indirizzo di puntatori
 - 12.03. Mancanza di () per una funzione
 - 12.04. Indici di array
 - 12.05. Array di caratteri e puntatori
 - 12.06. C è case-sensitive
 - 12.7. ";" chiude ogni istruzione

 13. Puntatori
 - 13.01. Cos'è un puntatore
 - 13.02. Puntatori e funzioni

- 13.03. Puntatori e array
- 13.04. Array di puntatori
- 13.05. Array multidimensionali e puntatori
- 13.06. Inizializzazione statica degli array di puntatori
- 13.07. Puntatori e strutture
- 13.08. Le "trappole" più comuni dei puntatori
 - 13.08.01. Non assegnare un puntatore ad un indirizzo di memoria prima di utilizzarlo
 - 13.08.02. Assegnazione indiretta illegale
- 14. Allocazione dinamica della memoria
 - 14.01. Malloc
 - 14.02. Linked Lists
- 15. Input ed output
 - 15.01. Streams
 - 15.01.01. Streams predefinite
 - 15.01.01.01. Redirezione
 - 15.02. Funzioni comuni di I/O
 - 15.03. Formattazione di I/O
 - 15.03.01. Printf
 - 15.04. Scanf
 - 15.05. Files
 - 15.05.01. Lettura e scrittura su files
 - 15.06. Sprintf ed Sscanf
 - 15.07. Input dalla linea di comando
 - 15.08. I/O di basso livello
- 16. Il preprocessore C
 - 16.01. #define
 - 16.02. #undef
 - 16.03. #include
 - 16.04. #if - Inclusione condizionale
- 17. Scrittura di grossi programmi
 - 17.01. File header
 - 17.02. Variabili esterne e funzioni
 - 17.02.01. Scopo delle variabili esterne
 - 17.03. L'utility Make
 - 17.04. Programmazione di Make
 - 17.05. Creazione di un makefile
 - 17.06. Macro di Make
 - 17.7. Esecuzione di Make

- 18. UNIX e il C
 - 18.01. Vantaggi di usare UNIX con il C
 - 18.02. Utilizzo delle chiamate di sistema UNIX e delle funzioni di libreria

- 18.03. Trattamento di file e directory
 - 18.03.01. Funzioni di trattamento delle directory
 - 18.03.02. Routine di trattamento dei file
 - 18.03.03. errno
- 18.04. Controllo e gestione dei processi
 - 18.04.01. Esecuzione di comandi UNIX da C
 - 18.04.01.01. execl()
 - 18.04.01.02. fork()
 - 18.04.01.03. wait()
 - 18.04.01.04. exit()
 - 18.04.02. Utilizzo di pipe in un programma C
 - 18.04.02.01. popen() - Piping formattato
 - 18.04.02.02. pipe() - Piping di basso livello
 - 18.04.03. Interruzioni e segnali
 - 18.04.03.01. Invio di segnali - kill()
 - 18.04.03.02. Ricezione di segnali - signal()
- 18.05. Times Up!!
- 19. Opzioni comuni del compilatore C
 - 19.01. Opzioni di compilazione
- 20. Funzioni della libreria standard C
 - 20.01. Manipolazione dei buffer
 - 20.02. Classificazione dei caratteri e conversione
 - 20.03. Conversione dei dati
 - 20.04. Manipolazione delle directory
 - 20.05. Manipolazione dei file
 - 20.06. Input e Output
 - 20.06.01. Stream I/O
 - 20.06.02. I/O di basso livello
 - 20.07. Matematica
 - 20.08. Allocazione di memoria
 - 20.09. Controllo dei processi
 - 20.10. Ricerca e ordinamento
 - 20.11. Manipolazione di stringhe
 - 20.12. Time

Qui di seguito verranno elencate brevemente alcune delle caratteristiche del C che definiscono il linguaggio stesso e che hanno contribuito alla popolarità che ha raggiunto come linguaggio di programmazione:

- dimensioni ridotte
- utilizzo frequente di chiamate a funzioni
- loose typing (a differenza del Pascal)
- linguaggio strutturato
- programmazione a basso livello facilmente disponibile
- implementazione dei puntatori (ampio uso di puntatori per memoria, vettori, strutture e funzioni)

Il C è ora diventato un linguaggio professionale ampiamente utilizzato per varie ragioni:

- ha strutture di alto livello
- può maneggiare attività di basso livello
- produce programmi efficienti
- può essere compilato su un'ampia gamma di computers

Il suo principale inconveniente è quello di avere un metodo scadente per l'identificazione di errori, che può escluderne l'utilizzo ai principianti. Comunque con un minimo di diligenza si può risolvere elegantemente questo problema, in quanto si possono violare le regole del C non appena si sono imparate (non molti linguaggi lo permettono). Nel caso in cui venga fatto correttamente e con attenzione, questo porta a sfruttare le potenzialità della programmazione C. Lo standard per i programmi C in origine era dato dalle caratteristiche messe a punto da Brian Kernighan. Al fine di rendere il linguaggio più accettabile a livello internazionale, venne messo a punto uno standard internazionale chiamato ANSI C (American National Standards Institute).

02. Storia del linguaggio C

Le pietre miliari nel corso dell'evoluzione del C come linguaggio sono elencate di seguito:

- UNIX developed c. 1969 - DEC PDP-7 Assembly Language
- BCPL - un OS facilmente accessibile che fornisce potenti strumenti di sviluppo prodotti a partire da BCPL. Si tratta di un assembler noioso, lungo ed incline agli errori
- Un nuovo linguaggio "B" come secondo tentativo c. 1970
- Un linguaggio "C" totalmente nuovo come successore di "B" c. 1971
- Dal 1973 UNIX OS, quasi totalmente scritto in C

03. Primo approccio

Un minimo programma in C è:

```
main()
{

}
```

che corrisponde a un programma in Pascal:

```
program minimum;

begin

end
```

Ogni programma C deve contenere una e una sola funzione main(). Per ogni parentesi graffa aperta (che corrisponde al begin in pascal) deve essercene una chiusa (che corrisponde all'end in pascal). I commenti possono essere posti ovunque utilizzando /* (inizio commento) e */ (fine commento), ma non si può inserire un commento in un altro. Ad esempio:

```
/* Esempio di programma in C */
main()
{
  /* Un ulteriore commento */           ESATTO
  /* Commento /* Ancora un commento */ /* ERRATO
}
```

Il seguente esempio è un programma che produce l'output sullo schermo della frase "Hello World":

```
main()
{
  printf("Hello World \n");
  exit(0);
}
```

L'istruzione "printf" è una funzione C che visualizza ciò che gli viene passato come argomento. Per creare un file contenente uno dei precedenti programmi si può utilizzare un qualsiasi text editor disponibile sulla macchina (vi, emacs, xedit, ...). Il nome del file deve avere l'estensione ".c", cioè chiamarsi, ad esempio, prog.c. Il contenuto, ovviamente, deve rispettare la sintassi C; per quanto riguarda gli esempi sopra riportati, potrebbero iniziare con una riga del tipo:

```
/* Esempio ... */      (anche con una linea vuota che la precede)
```

e terminare con la linea

```
} /* Fine del programma */ (anche con una linea vuota che la segue)
```

04. Compilazione di un programma C

Per compilare il programma si utilizza il comando cc seguito dal nome del programma C sorgente, dove "cc" è il nome del compilatore C. Ad esempio:

cc prog.c

Se il compilatore trova errori (in genere syntax error, come errori di battitura, errori di sintassi delle parole chiave o ";" omessi), questi vengono identificati e visualizzati; in caso contrario, viene creato il file eseguibile a.out. Il compilatore non identifica eventuali errori di logica del programma, quindi potrebbero essere eseguite delle operazioni errate, ed è compito dell'utente trovarle (anche con l'ausilio di appositi programmi di debugging). In fase di compilazione possono essere specificate anche ulteriori opzioni: la più utilizzata è "-o nome-file", che crea l'eseguibile con il nome nome-file invece di a.out, ma ne esistono altre come ad esempio "-c" (opzione senza argomenti, per la soppressione di link). Altra opzione possibile è "-g", con cui è necessario compilare per poter utilizzare il debugger "dbx". Ad esempio:

```
cc prog.c -o prog (oppure cc -o prog.c prog)
cc -c prog.c -o prog
cc -g prog.c -o prog
```

Per far eseguire il programma è sufficiente scrivere il nome dell'eseguibile creato (è ovvio che il file eseguibile deve avere i permessi per l'esecuzione, solitamente assegnati automaticamente in fase di compilazione): si avranno visualizzati sullo schermo gli eventuali risultati. Nel momento dell'esecuzione è possibile osservare ed identificare eventuali errori di run-time, come ad esempio le divisioni per zero; in tal caso l'esecuzione termina irregolarmente e viene generato un file core con lo stato del programma in esecuzione al momento del verificarsi dell'errore. Se il programma in esecuzione non rilascia errori ma produce output errati, è evidente che contiene errori logici; questi andranno corretti editando il programma sorgente, questo dovrà essere ricompilato e si potrà lanciare nuovamente l'esecuzione.

La compilazione del programma C avviene attraverso le seguenti fasi:

- un preprocessore che accetta il codice sorgente come input ed è responsabile della:

- rimozione di commenti
- interpretazione di speciali direttive per il preprocessore denotate da "#".

Ad esempio:

```
#include - include il contenuto di un determinato file (solitamente chiamato header, con suffisso ".h");
```

```
#include <math.h> - standard library maths file.
```

```
#include <stdio.h> - standard library I/O file
```

```
#define - definisce un nome simbolico o una costante (sostituzione di una macro):
```

```
#define MAX_ARRAY_SIZE 100
```

- il compilatore C che traduce il codice sorgente ricevuto dal preprocessore in codice assembly.

- l' assembler che crea il codice oggetto (in UNIX i file con il suffisso o sono i file in codice oggetto, che corrispondono ai file .obj in MSDOS).

- il link editor che combina le funzioni definite in altri file sorgenti o definite in librerie, con la funzione main() per creare il file eseguibile.

Infatti molte delle funzioni presenti in altri linguaggi non sono incluse nel C (ad esempio, funzioni di I/O, di manipolazione di stringhe o matematiche), ma il C fornisce tali funzionalità attraverso un ricco insieme di librerie di funzioni. Molte applicazioni C includono librerie standard di funzioni per coprire le utilità mancanti. In questa fase vengono anche ricostruiti i riferimenti alle variabili esterne utilizzate nei sorgenti C.

05. Struttura di un programma C

Un programma C ha in linea di principio la seguente forma:

Comandi per il preprocessore

Definizione di tipi
Prototipi di funzioni (dichiarazione dei tipi delle funzioni e delle
variabili passate alle funzioni)
Variabili
Funzioni

Vediamo l'esempio di un programma:

```
main()
{
printf("I like C\n");
exit(0);
}
```

Note:

- Il C richiede un punto e virgola alla fine di ogni statement.
- printf() è una funzione standard richiamata da main.
- \n significa una nuova linea (a capo).
- exit() è anch'essa una funzione standard che fa terminare il programma (qui non sarebbe necessaria in quanto è l'ultima linea di main e il programma terminerebbe comunque).

=====
Variabili
=====06.

Il C ha i seguenti tipi di dati:

Tipo	Size (byte)
char	1
unsigned char	1
short int	2
unsigned short int	2
(long) int	4
float	4
double	8

Sui sistemi UNIX tutte le variabili dichiarate "int" sono considerate "long int", mentre "short int" deve essere dichiarato esplicitamente. È importante notare che in C non esiste un tipo di variabile booleano, quindi si possono utilizzare variabili "char", "int" o meglio "unsigned char". "unsigned" può essere utilizzato con tutti i tipi "char" e "int".

Per dichiarare una variabile si scrive:

```
var_tipo elenco-variabili-separate-da-virgole ;
```

Le variabili globali si definiscono al di sopra della funzione main(), nel seguente modo:

```
short number,sum;
int bignumber,bigsum;
char letter;
main()
{
...
}
```

È possibile preinizializzare una variabile utilizzando = (operatore di assegnazione). Ad esempio:

```
int i,j,k=1;
float x=2.6,y;
char a;
```

Vediamo due esempi di inizializzazione di variabili che si equivalgono, senza però dimenticare che il metodo utilizzato nel primo esempio risulta più efficiente:

Esempio 1: `float sum=0.0;`
`int bigsum=0;`
`char letter='À';`
`main()`
`{`
`...`
`}`

Esempio2: `float sum;`
`int bigsum;`
`char letter;`
`main()`
`{`
`sum=0.0;`
`bigsum=0;`
`letter='À';`
`...`
`}`

È possibile effettuare assegnazioni multiple purchè le variabili siano dello stesso tipo. Ad esempio:

```
int somma;
char letter="A";
main()
{
  int a,b,c=3;
  a=b=c;
}
```

dove l'istruzione `a=b=c` (con `c=3`) corrisponde ad `a=3`, `b=3` e `c=3`, ma anche in questo caso risulta più efficiente il primo metodo.

Si possono definire nuovi propri tipi di variabili utilizzando "typedef" (questo risulta utile quando si creano strutture complesse di dati). Come esempio di utilizzo semplice consideriamo come sia possibile creare i due nuovi tipi di variabile "real" e "letter", che potranno successivamente essere utilizzati alla stessa maniera dei tipi predefiniti del C.

Ad esempio:

```
typedef float real;
typedef char letter;
variabili dichiarate:
real sum=0.0;
letter nextletter;
```

06.01. Stampa ed input di variabili

Il C sfrutta l'output formattato. Per stampare il contenuto di una variabile si utilizza la funzione `printf()`. Bisogna però specificare il formato della variabile utilizzando il carattere speciale di formattazione "%" seguito dal carattere che definisce un certo formato per una variabile:

```
%c - char
%d - int
```

%f - float

Ad esempio: `printf("%c%d%f",letter,somma,z);`

Nota: l'istruzione di formattazione è racchiusa tra "", e le variabili vengono espresse di seguito; assicurarsi che l'ordine dei formati ed il tipo di dato delle variabili coincidano.

Sempre a proposito della funzione "printf", vediamo il seguente esempio di una istruzione di stampa:

```
printf(".\n.1\n..2\n...3\n");
```

per la quale l'output sarà:

```
.
.1
..2
...3
```

`scanf()` è la funzione per l'input di valori a strutture di dati. Il suo formato è simile a quello di `printf()`:

```
scanf("%c%d%f",&ch,&i,&x);
```

Nota: "&" si riferisce all'indirizzo della variabile, e va sempre messo davanti ai nomi di variabili in acquisizione; il motivo verrà spiegato nel paragrafo dei "puntatori".

07. Operatori

07.1. Operatori aritmetici

Come già accennato, le assegnazioni in C vengono effettuate utilizzando "=". Oltre agli operatori aritmetici standard +, -, *, / e all'operatore % (modulo) per gli interi, in C si hanno anche gli operatori incremento ++ e decremento --, che possono essere preposti o posposti all'argomento. Se sono preposti il valore è calcolato prima che l'espressione sia valutata, mentre se sono posposti il valore viene calcolato dopo la valutazione della espressione.

Ad esempio:

```
1)   int x,z=2;
      x=(++z)-1;
```

A questo punto $x=2$ e $z=3$

```
2)   int x,z=2;
      x=(z++)-1;
```

A questo punto $x=1$ e $z=3$

Riportiamo un ulteriore esempio:

```
int x,y,w;
main()
{
  x=((++z)-(w--))%100;
}
```

che equivale alle seguenti istruzioni:

```
int x,y,w;
```

```

main()
{
z++;
x=(z-w)%100;
w--;
}

```

È importante sottolineare che un'istruzione del tipo `x++` è più veloce della corrispondente `x=x+1`.

L'operatore "%" (modulo) può essere utilizzato solamente con le variabili di tipo integer; la divisione "/" è utilizzata sia per gli integer che per i float.

A proposito della divisione riportiamo un altro esempio:

```
z=3/2
```

dove `z` avrà valore 1, anche se è stato dichiarato come float (di regola, se entrambi gli argomenti della divisione sono integer, allora verrà effettuata una divisione integer); per avere un risultato corretto sarà necessario scrivere:

```

z=3.0/2 oppure
z=3/2.0 o, ancora meglio,
z=3.0/2.0

```

Inoltre esiste una forma contratta per espressioni del tipo:

```
expr1 = expr1 op expr2
```

(ad esempio: `i=i+2` oppure `x=x*(y+3)`) che diventano:

```
expr1 op = expr2
```

Per cui `i=i+2` può essere scritta nel modo contratto come `i+=2` od `x=x*(y+3)` diventare `x*=y+3`.

Nota: l'espressione `x*=y+3` corrisponde a `x=x*(y+3)` e non a `x=x*y+3`.

07.02. Operatori di confronto

Per testare l'uguaglianza si usa "==" mentre per la disuguaglianza "!=". Ci sono poi gli operatori "<" (minore), ">" (maggiore), "<=" (minore o uguale), ">=" (maggiore o uguale).

NB. `if (i==j) ...` esegue il contenuto dell'`if` se `i` è uguale a `j`, ma `if (i=j) ...` è ancora sintatticamente esatto ma effettua l'assegnazione del valore di `j` ad `i` e procede se `j` è diverso da zero in quanto viene interpretato il valore TRUE (è come scrivere `if i ...` con `i` diverso da zero). In questo caso si tratterebbe di una "assegnazione di valore", una caratteristica chiave del C.

07.03. Operatori logici

Gli operatori logici, solitamente utilizzati con le istruzioni condizionali che vedremo più avanti, sono "&&" per AND logico e "||" per OR logico.

Nota: "&" e "||" hanno un significato diverso, poichè sono bitwise AND e OR.

07.4. Operatori di basso livello

Nel capitolo relativo ai puntatori si vedrà come questi permettano il controllo delle operazioni di memoria di basso livello. Molti programmi (in particolare le applicazioni di gestione del sistema) devono realmente operare a basso livello, poichè lavorano su bytes individuali. È importante notare che la combinazione di puntatori e di operatori bit-level rendono il C utilizzabile per molte applicazioni a basso livello e possono quasi sempre sostituire il codice assembly (ricordiamo che solamente circa il 10% di UNIX è un codice assembly, mentre il resto è C).

07.04.01. Operatori di bitwise

Gli operatori di bitwise (che operano sui singoli bit) sono i seguenti:

- "&" AND
- "||" OR
- "^" XOR
- "~" Complemento a 1 (0=>1, 1=>0)
- "<<" shift a sinistra
- ">>" shift a destra

Nota: fare attenzione, come già detto in precedenza, a non confondere & con && (& è "bitwise and", mentre && è "logical and"); la stessa cosa vale per | e ||.

"~" è un operatore unario, cioè opera su un solo argomento indicato a destra dell'operatore.

Gli operatori di shift eseguono un appropriato shift dall'operatore indicato a destra a quello indicato a sinistra. L'operatore destro deve essere positivo. I bits liberati vengono riempiti con zero (cioè non si tratta di una rotazione, con recupero sul lato opposto dei bit shiftati).

Ad esempio: `z<<2` shifta i bit in `z` di due posti verso sinistra:

così, se `z=00000010` (binario) o 2 (decimale)
allora, `z>>=2` => `z=00000000` (binario) o 0 (decimale)
inoltre, `z<<=2` => `z=00001000` (binario) o 8 (decimale)

Quindi, uno shift a sinistra è equivalente ad una moltiplicazione per 2; similmente, uno shift a destra equivale ad una divisione per 2.

Nota: l'operazione di shift è molto più veloce della reale moltiplicazione (*) o divisione (/); così, se occorrono veloci moltiplicazioni o divisioni per 2 si può utilizzare lo shift.

Per illustrare le molteplici caratteristiche degli operatori di bitwise, riportiamo una funzione (`bitcount`) che somma 2 bit settati ad 1 un un numero ad 8 bit (`unsigned char`) passato come argomento alla funzione:

```
int bitcount(unsigned char x)
{
    int count;
    for (count=0; x!=0; x>>=1);
        if (x&01)
            count++;
    return count;
}
```

Questa funzione mostra molti punti del programma C:

- il loop "for" non viene usato per semplici operazioni di somma
- $x \gg= 1 \Rightarrow x = x \gg 1$
- il loop "for" shifta ripetutamente a destra x, finchè x diventa 0
- il controllo "if" utilizza la valutazione dell'espressione $x \& 01$
- $x \& 01$ controlla il primo bit di x, ed esegue `count++` se questo è 1

07.04.02. Bit Fields

I Bit Fields permettono il raggruppamento dei dati in una struttura. Questa tecnica viene usata soprattutto quando la gestione della memoria o la memorizzazione dei dati sono una meta molto ambita.

Tipici esempi sono costituiti da:

- raggruppamento di parecchi oggetti in una parola (i flag di un bit possono essere compattati); ad esempio, la tabella dei simboli nell'ambito dei compilatori;
- lettura di formati di file esterni (formati di file non standard possono essere importati); ad esempio, gli interi di 9 bit.

Il C permette di fare questo in una definizione di struttura, mettendo ":lunghezza-bit" dopo la variabile stessa, e cioè:

```
struct packed-struct {
    unsigned int f1:1;
    unsigned int f2:1;
    unsigned int f3:1;
    unsigned int f4:1;
    unsigned int type:4;
    unsigned int funny_int:9;
} pack;
```

Qui la struttura packed-struct contiene 6 elementi: 4 flag da 1 bit (f1,... f4) e funny_int da 9 bit.

Il C automaticamente raggruppa assieme i campi di bit elencati nell'esempio appena riportato.

Solitamente si accede ai membri della struttura nel seguente modo:

```
pack.type = 7;
```

Notiamo che:

- solamente "n" bit di basso livello possono essere assegnati ad un numero di "n" bit. Così il campo "type" non può assumere valori maggiori di 15 (4 bits long);
- i bit fields vengono sempre convertiti al tipo intero prima di eseguirvi delle operazioni;
- è permesso "mescolare" tipi normali con bit fields;
- la definizione di "unsigned" è importante, per assicurarsi che per i flags non venga usato nessun bit per il segno.

07.05. Ordine di precedenza degli operatori

È necessario fare attenzione al significato di un'espressione come $a + b * c$, dove potremmo volere sia l'effetto di $(a + b) * c$ sia quello di $a + (b * c)$. Tutti gli operatori hanno una propria priorità, e gli operatori ad alta priorità sono valutati prima di quelli a bassa priorità. Gli operatori con la stessa priorità sono valutati da sinistra a destra: così $a - b - c$ è valutato $(a - b) - c$, come ci si può aspettare.

L'ordine di priorità, dalla più alta alla più bassa, degli operatori in C è:

```
()[]->.
!~*& sizeof cast ++ --
```

(these are right -> left)

*/%

+-

< <= >= >

== !=

&

^

|

&&

||

?: (right -> left)

= += -= (right -> left)

,(comma)

Quindi:

```
"a < 10 && 2 * b < c"
```

è interpretato come:

```
"(a < 10) && ((2 * b) < c)".
```

ed anche:

```
a=
```

```
    b=
```

```
        spokes / spokes_per_wheel
```

```
        + spares;
```

è valutato come:

```
a=
```

```
    (b=
```

```
        (spokes / spokes_per_wheel)
```

```
        + spares
```

```
    );
```

08. Strutture di controllo

Quelli che seguono sono i vari metodi con cui il C può controllare il flusso logico di un programma. A parte alcune minime differenze sintattiche, queste istruzioni sono simili a quelle che si possono trovare negli altri linguaggi. Come abbiamo visto, in C esistono le seguenti operazioni logiche:

==, !=, ||, &&.

Un altro operatore è il not "!" unario (ha un solo argomento).

Questi operatori sono utilizzati congiuntamente alle istruzioni di seguito riportate.

08.01. If

L'istruzione "if" ha le stesse funzioni degli altri linguaggi. Può avere tre forme di base:

```
if (expression)
    statement
```

```
if (expression)
    statement1
```

```
else
  statement2
```

```
if (expression1)
  statement1
else if (expression2)
  statement2
else
  statement3
```

Ad esempio:

```
int x,y,z;
main()
{
  int w;
  ...
  if (x<0)
  {
    z=w;
    ...
  }
else
{
  z=y;
  ...
}
}
```

08.02. Operatore "?"

L'operatore ? (ternary condition) è la forma più efficiente per esprimere semplici if statements. Ha la seguente forma:

expression1 ? expression2 : expression3

che equivale a:

if expression1 then expression2 else expression3

Ad esempio:

`z=(a>b) ? a : b`

cioè

```
if (a>b)
    z=a;
else
    z=b;
```

assegna a z il massimo tra a e b.

08.03. Switch

Permette scelte multiple tra un insieme di items. La sua forma generale è:

```
switch (expression) {
    case item1:
        statement1;
        break;
    case item2:
        statement2;
        break;
    .
    .
    .
    case itemn:
        statementn;
        break;
    case default:
        statement;
        break;
}
```

Il valore degli item deve essere una costante (le variabili non sono permesse). Il `break` serve per terminare lo `switch` dopo l'esecuzione di una scelta, altrimenti verrà valutato anche il caso successivo (questo, a differenza di molti altri linguaggi).

È possibile anche avere un'istruzione nulla, includendo solamente un ";" oppure lasciando fallire l'istruzione di `switch` omettendo qualsiasi frase (come nell'esempio di seguito). Il caso "default" è facoltativo e raggruppa tutti gli altri casi.

Ad esempio:

```
switch (letter) {
    case 'À':
    case 'È':
    case 'Ì':
    case 'Ò':
    case 'Ù':
        numerovocali++;
        break;
    case " ":
        numerospazi++;
        break;
    default:
        numerocostanti++;
}
```

```
    break;
}
```

In questo caso se letter è una vocale ('À','È','Ì','Ò','Ù) viene incrementato il valore della variabile numerovocali, se è uno spazio (" ") si incrementa numerospazi e altrimenti (se nessuno dei casi precedenti è vero) viene eseguita la condizione di default e quindi viene incrementato numerocostanti.

08.04. For

L'istruzione C "for" ha la seguente forma:

```
for (expression1; expression2; expression3)
    statement;
    {or block of statements}
```

dove expression1 inizializza, expression2 è il test di termine e expression3 è il modificatore (che può anche essere più di un semplice incremento). Nota: fondamentalmente il C tratta le istruzioni "for" come i cicli di tipo "while".

Ad esempio:

```
int x;
main()
{
    for (i=0;i<3;i++)
        printf("x=%d\n",x);
}
```

che genera come output sullo schermo:

```
x=0
x=1
x=2
```

Gli esempi che seguono sono tre forme valide delle istruzioni "for" in C:

```
for (x=0;((x<3)&&(x>9));x++)    for (x=0,y=4;((x<3)&&(y>9));x++,y+=2)
```

in cui si può notare che le espressioni multiple possono essere separate da una ",";

```
for (x=0,y=4,z=4000;z/=10)
```

in cui si può notare che il loop continua l'iterazione fino a quanto z diventa 0.

08.05. While

L'istruzione "while" ha la seguente forma:

```
while (expression)
    statement;
```

Ad esempio:

```
int x=3;
main()
{
    while (x>0)
    {
        printf("x=%d\n",x);
        x--;
    }
}
```

```
    }  
}
```

che genera come output sullo schermo:

```
x=3  
x=2  
x=1
```

While può accettare non solo condizioni ma anche espressioni, per cui risultano corrette le seguenti istruzioni:

```
while (x-);  
while (x=x+1);  
while (x+=5);
```

Utilizzando questo tipo di espressioni, solo quando il risultato di $x-$, $x=x+1$ oppure $x+=5$ ha valore 0 la condizione di `while` fallisce e si esce dal loop.

È possibile avere anche complete operazioni di esecuzione nelle espressioni "while":

```
while (i++<10)
```

che incrementa `i` fino a raggiungere il valore 10;

```
while ((ch=getchar())!='q')  
    putchar(ch);
```

che usa le funzioni `getchar()` e `putchar()` delle librerie standard, che rispettivamente leggono un carattere dalla tastiera e scrivono un determinato carattere sullo schermo.

Il loop `while` continua a leggere dalla tastiera e a visualizzare sullo schermo il carattere digitato, fino a quando non venga battuto il carattere "q".

08.06. Do-While

L'istruzione C "do-while" ha la seguente forma:

```
do  
    statement;  
while (expression);
```

(è simile al Pascal `repeat ... until`, eccetto il fatto che l'espressione dell'istruzione `do-while` è vera)

Ad esempio:

```
int x=3;  
main()  
{  
    do {  
        printf("x=%d\n",x-); /* le graffe sono superflue, visto */  
    } while (x>0); /* che racchiudono solamente una */  
}
```

```

        }
while (x>0);
}

```

Il cui output è:

```

x=3
x=2
x=1

```

Nota: l'operatore finale "x-" indica che viene usato il valore corrente di x mentre stampa, e poi viene decrementato x.

08.07. Break e Continue

Il C fornisce due comandi per controllare i loop:

`break` - esce da un loop o da uno switch

`continue` - salta una iterazione del loop

Consideriamo il seguente esempio, dove leggiamo un valore integer e lo elaboriamo in accordo con le seguenti condizioni. Se il valore che abbiamo letto è negativo, dovremo stampare un messaggio di errore ed abbandonare il loop. Se il valore letto è maggiore di 100, dovremo ignorarlo e continuare con il successivo valore in input. Se il valore è 0, dovremo terminare il loop.

```

/* Viene letto un valore intero ed elaborato purchè sia maggiore di 0 e minore di 100 */
while (scanf("%d",&value) == 1 && value !=0) {
    if (value<0) {
        printf("Valore non ammesso\n");
        break; /* Abbandona il loop */
    }
    if (value>100) {
        printf("Valore non ammesso\n");
        continue; /*Torna nuovamente all'inizio del loop */
    }
    /*Elabora il valore letto*/
    /*che è sicuramente tra 0 e 100 */
    :
    .
}

```

09. Arrays

09.01. Array singoli e multidimensionali

Un esempio di definizione di un array in C è :

```
int elenco_numeri[50];
```

e si accede agli elementi dell'array nel seguente modo:

```
terzo_numero= elenco_numeri[2];
elenco_numeri[5]=100;
```

NB. In C gli Array subscripts iniziano da 0 e finiscono alla dimensione dell'array meno uno. Nell'esempio precedente il range è 0-49, cioè `elenco_numeri` è un array di 50 elementi e si ha:

```
elenco_numeri[0],elenco_numeri[1],...elenco_numeri[49].
```

Questa è una grossa differenza fra il C e gli altri linguaggi e richiede un po' di pratica per raggiungere " la giusta disposizione d'animo".

Array multidimensionali sono così definiti:

```
int tabella_numeri[50][50]    => per due dimensioni
int big_D[20][30][10][40]   => per più di due dimensioni
```

e si accede agli elementi nel seguente modo:

```
numero=tabella_numeri[5][32];
tabella_numeri[1][23]=100;
```

09.02. Stringhe

In C le stringhe sono definite come array di caratteri. Ad esempio, la seguente istruzione definisce una stringa di 50 caratteri:

```
char name[50];
```

Il C non ha però un sistema maneggevole per costruire le stringhe, così le seguenti assegnazioni non sono valide:

```
char firstname[50], lastname[50], fullname[50];

firstname = "Mario"           /* illegale */
lastname  = "Rossi"          /* illegale */
fullname  = "Sig."+firstname+lastname /* illegale */
```

Esiste però una libreria di routines per il trattamento delle stringhe ("`< string.h >`"). Per maneggiare le stringhe si possono usare puntatori ad array di char (come vedremo più avanti). Per stampare una stringa si usa `printf()` con lo speciale carattere di controllo `%s`:

```
printf("%s", nome);  Nota: è sufficiente avere il nome della stringa.
```

Al fine di permettere l'utilizzo di stringhe con lunghezza variabile, il carattere `\0` viene utilizzato per indicare la fine di una stringa. In questo modo, se abbiamo una stringa dichiarata di 50 caratteri (`char name[50];`), e la utilizziamo per memorizzare il nome "Dave", il suo contenuto (a partire da sinistra) sarà la parola Dave immediatamente seguita dal segno di fine stringa `\0`, e quindi tutti gli altri caratteri (fino ad arrivare alla lunghezza di 50) risulteranno vuoti.

10. Funzioni

Il C fornisce delle funzioni anch'esse simili alla maggior parte degli altri linguaggi. Una differenza è che il C considera "`main()`" come una funzione. A differenza di alcuni linguaggi, come il Pascal, il C non ha procedure poiché usa le funzioni per soddisfare entrambe le esigenze.

La forma generale di una funzione è:

```
returntype function_name (parameterdef1, parameterdef2, ...)
{
    local variables
    function code (C statements)
}
```

Se manca la definizione del tipo della funzione ("`returntype`", tipo della variabile di ritorno della funzione), il C assume che il ritorno della funzione è di tipo `integer`; questo può essere una delle cause di problemi nei programmi.

Esempio di una funzione che calcola la media tra due valori:

```
float calcolamedia(float a, float b)
{
    float media;
    media=(a+b)/2;
    return(media);
}
```

Per richiamare tale funzione si procede nel seguente modo:

```
main()
{
    float a=10, b=25, risultato;
    risultato=calcolamedia(a,b);
    printf("Valore medio= %f\n",risultato);
}
```

Nota: l'istruzione "return" ritorna il risultato della funzione al programma principale.

10.01. Funzioni "void"

Se non si vuole ritornare alcun valore da una funzione è sufficiente dichiararla di tipo void ed omettere il return. Ad esempio:

```
void quadrati()
{int loop;

    for (loop = 1; loop < 10; loop++);
    printf("%d\n",loop*loop);
}

main()
{quadrati()
}
```

Nota: è obbligatorio mettere le parentesi () dopo il nome della funzione anche se non ci sono parametri, a differenza di altri linguaggi.

10.02. Funzioni ed array

Possono essere passati alle funzioni come parametri anche array singoli o multidimensionali. Gli array monodimensionali possono essere passati nel seguente modo:

```
float trovamedia(int size,float list[])
{int i;
    float sum=0.0;

    for (i = 0; i < size; i++)
        sum+=list[i];
    return(sum/size);
}
```

In questo esempio la dichiarazione "float list[]" dichiara al C che "list" è un array di float. Non viene specificata la dimensione di un array quando è un parametro di una funzione.

Array multidimensionali possono essere passati alle funzioni nel seguente modo:

```
void stampatabella(int xsize, int ysize,float tabella[][5])
{int x,y;
```

```

for (x = 0; x < xsize; x++) {
    for (y = 0; y < ysize; y++)
        printf("%t%f" tabella[x][y]);
    printf("\n");
}
}

```

In questo esempio "float tabella[][5]" dichiara al C che tabella è un array di float di dimensioni Nx5. È importante notare che dobbiamo specificare la seconda dimensione (e le successive) del vettore, ma non la prima dimensione. Quindi, riepilogando, nel caso di array singoli non è necessario specificare la dimensione dell'array nella definizione come parametro della funzione, mentre nel caso di array multidimensionali si può non specificare solo la prima dimensione.

10.03. Prototipi di funzioni

Prima di usare una funzione, il C deve riconoscere il tipo di ritorno e il tipo dei parametri che la funzione si aspetta. Lo standard ANSI del C ha introdotto un nuovo e migliore metodo per fare questa dichiarazione rispetto alle precedenti versioni di C (ricordiamo che tutte le nuove versioni del C aderiscono ora allo standard ANSI). L'importanza della dichiarazione è doppia:

- viene fatta per avere un codice sorgente più strutturato e perciò facile da leggere ed interpretare;
- permette al compilatore C di controllare la sintassi delle chiamate di funzioni.

Il modo in cui questo viene fatto dipende dallo scopo della funzione. Fondamentalmente, se una funzione è stata definita prima di essere usata (call) allora è possibile semplicemente usare la funzione. Nel caso contrario, è obbligatorio dichiarare la funzione; la dichiarazione stabilisce in modo semplice il ritorno della funzione ed il tipo dei parametri utilizzati da questa. È buona norma (e solitamente viene fatto) dichiarare tutte le funzioni all'inizio del programma, sebbene non sia strettamente necessario.

Per dichiarare un prototipo di funzione bisogna semplicemente stabilire il ritorno della funzione, il nome della funzione e tra le parentesi elencare il tipo dei parametri nell'ordine in cui compaiono nella definizione di funzione.

Ad esempio:

```
int strlen(char[]);
```

Questo dichiara che una funzione di nome "strlen" ritorna un valore integer ed accetta una singola stringa come parametro.

Nota: le funzioni e le variabili possono essere dichiarate sulla stessa linea di codice sorgente. Questa procedura era molto più diffusa nei giorni del pre-ANSI C; da allora le funzioni solitamente vengono dichiarate separatamente all'inizio del programma. La prima procedura risulta ancora perfettamente valida, purchè venga rispettato l'ordine in cui gli oggetti compaiono nella definizione della funzione.

Ad esempio:

```
int length, strlen(char[]);
```

dove "length" è una variabile, e "strlen" è la funzione (come nell'esempio precedente).

11. Ulteriori tipi di dati

11.01. Strutture

Le strutture in C sono simili ai records in Pascal.

Ad esempio:

```

struct gun
{
    char name[50];
    int magazinesize;
    float calibre;
};

struct gun arnies;

```

Viene così definita una nuova struttura `gun` e definita `arnies` di tipo `struct gun`.

Nota: "gun" è un'etichetta (tag) per la struttura che serve come abbreviazione per le successive dichiarazioni. È necessario solamente dichiarare "struct gun" e il corpo della struttura è implicito come viene fatto per creare la struttura "arnies"; il tag è opzionale.

Le variabili possono anche essere dichiarate tra "}" e ";" di una dichiarazione di struttura; ad esempio:

```

struct gun
{
    char name[50];
    int magazinesize;
    float calibre;
} arnies;

```

che equivale al precedente esempio di definizione di una nuova variabile strutturata di nome "arnies".

Una struttura può essere pre-inizializzata al momento della dichiarazione:

```
struct gun arnies={"Uzi",30,7};
```

Per accedere ai membri (o campi) di una struttura il C fornisce l'operatore ".".

Ad esempio:

```
arnies.magazinesize=100;
```

Anche con le strutture si può utilizzare typedef. La seguente istruzione crea un nuovo tipo "agun" che è di tipo "struct gun" e può essere inizializzato come al solito:

```

typedef struct gun
{
    char name[50];
    int magazinesize;
    float calibre;
} agun;

agun arnies= {"Uzi",30,7};

```

Qui "gun" è ancora un'etichetta della struttura ed è opzionale; agun è un nuovo tipo di dato e arnies è una variabile di tipo agun (che è una struttura).

Il C permette anche la definizione array di strutture:

```
agun arniesguns[1000];
```

che si possono utilizzare nel seguente modo:

```
arniesguns[50].calibre=5;
```

dove il campo "calibre" del record 50 di arniesguns assume valore 5;

```
itscalibre= arniesguns[50].calibre;
```

dove viene assegnato alla variabile itscalibre il valore del campo calibre del record 50 di arniesguns.

11.02. Unioni

Un'unione è una variabile che può tenere (in momenti diversi) oggetti di diversa dimensione e tipo. Il C usa l'istruzione "union" per creare unioni; ad esempio:

```
union number
{
    short shortnumber;
    long longnumber;
    double doublenumber;
} anumber
```

In questo modo viene definita un'unione chiamata number e un riferimento ad essa chiamato anumber. "number" è un'etichetta (tag) di unione e funziona alla stessa maniera del tag delle strutture. Si accede ai membri dell'unione come per i membri delle strutture.

Ad esempio:

```
printf("%ld\n",anumber.longnumber);
```

Questa istruzione visualizza semplicemente il valore di longnumber.

Quando il compilatore C alloca la memoria per le unioni, riserva sempre lo spazio necessario per il membro più grande (nell'esempio sopra riportato, sono 8 bytes per il tipo "double").

Per fare sì che il programma possa tenere traccia del tipo della variabile di unione usata in un determinato momento, è diffusa l'abitudine di avere una struttura (con registrate le unioni) e una variabile che indica il tipo dell'unione.

Ad esempio:

```
typedef struct
{int maxpassengers;
} jet;

typedef struct
{int liftcapacity;
} helicopter;

typedef struct
{int maxpayload;
} cargoplane;

typedef struct
{jet jetu;
helicopter helicopteru;
cargoplane cargoplaneu;
} aircraft;

typedef struct
{aircrafttype kind;
int speed;
aircraft description;
} an_aircraft;
```

Questo esempio definisce una unione di base aircraft, che può essere sia jet, helicopter o cargoplane. Nella struttura an_aircraft c'è un tipo di elemento che indica quale struttura è contenuta in quel momento.

11.03. Type-casting

Il C è uno dei pochi linguaggi che permette la coercizione, e cioè permette di forzare una variabile di un tipo ad essere una variabile di un altro tipo utilizzando l'operatore "()".

Ad esempio:

```
int numerointero;
int numerointero2=10;
float numerofloat=6.34;
float numerofloat2;
char lettera='À';

numerointero=(int)numerofloat; /* assegna il valore 6 (parte intera) */
numerointero=(int)lettera; /* assegna il valore 65 (codice ASCII)*/
numerofloat2=(float)numerointero2 /* assegna 10.0 (valore float) */
```

Alcuni type-casting vengono fatti automaticamente, principalmente in relazione alle capacità dei numeri integer. È buona regola eseguire il type-casting tutte le volte che si è in dubbio sulla corrispondenza degli operatori nelle assegnazioni.

Altro uso che ne viene fatto è all'interno delle divisioni, per assicurarsi che dia il risultato voluto; se abbiamo due numeri integer come operatori e vogliamo che il risultato sia un float, allora dovremo agire come segue:

```
int intnumber,anotherint;
float floatnumber;
floatnumber=(float)intnumber/(float)anotherint
```

Questa operazione assicura una divisione in floating-point.

11.04. Enumerated Types

Gli enumerated types contengono un elenco di costanti che possono essere indirizzate con valori integer. Per dichiarare tali tipi si utilizza "enum"; vengono dichiarati i tipi e le variabili come nell'esempio che segue:

```
enum colori {rosso, giallo, verde, blu} pennarello;
enum giorni {lun,mar,mer,gio,ven,sab,dom} settimana;
enum colori pulsante, nastro;
```

In tale esempio viene dichiarato colori come enumerated type e la variabile pennarello con 4 valori accettabili definiti, mentre la variabile settimana di tipo giorni ha 7 valori accettabili definiti. Le variabili pulsante e nastro sono di tipo colori.

Ogni item nell'elenco di valori accettabili è detto enumeration constant. Il C mappa ogni enumeration constant ad un'intero, per cui è ad esempio possibile scrivere:

```
settimana=verde;
```

che come risultato fa sì che settimana abbia valore 2, perchè di default a ogni membro dell'elenco di variabili è assegnato un valore incrementale partendo da 0 per il primo valore (come già visto per gli array).

È possibile definire valori diversi agli elementi:

```
enum colori {rosso=10, giallo=30, verde, blu=giallo};
```

Un ulteriore esempio relativo all'assegnazione di valori diversi è il seguente:

```
enum escapes {bell='\à',
              backspace='\b', tab='\t',
              newline='\n', vtab='\v',
              return='\r'};
```

È anche possibile annullare il valore iniziale 0:

```
enum months (jan=1,feb,mar,...dec);
```

dove è implicito che febbraio=2, marzo=3 e così via.

11.05. Variabili statiche

Una variabile statica è locale ad una particolare funzione. È inizializzata una sola volta, la prima volta che tale funzione viene chiamata e il suo valore resta inalterato quando si esce dalla funzione, per cui quando si richiama nuovamente la funzione tale variabile ha ancora il valore assegnatogli precedentemente.

Per definire statica una variabile è sufficiente anteporre la parola `static` alla dichiarazione della variabile.

Ad esempio:

```
void stat(); /* prototype function */

main()
{int i;

  for (i=0; i<5; ++i)
    stat()
  }

stat()
{int auto_var=0;
 static int static_var=0;

  printf("auto=%d, static=%d \n", auto_var, static_var);

  ++ auto_var;
```

```
    ++ static_var;
}
```

Il cui output sarà:

```
auto_var=0, static_var=0
auto_var=0, static_var=1
auto_var=0, static_var=2
auto_var=0, static_var=3
auto_var=0, static_var=4
```

La variabile `auto_var` viene creata ogni volta, mentre la variabile `static_var` è creata una sola volta ed il suo valore memorizzato.

12. Errori comuni in C

Prima di procedere con l'analisi delle caratteristiche più avanzate del C, è importante analizzare le cause dei possibili errori nei programmi C. Alcuni di questi errori vengono fatti facilmente, alcuni altri vengono fatti nel caso in cui si conosca un linguaggio con una sintassi a volte simile al C (come ad esempio il Pascal)

12.01. Assegnazione (=) al posto di confronto (==)

Il C utilizza l'assegnazione per valore, così:

```
if (a=b)
```

è sintatticamente corretta.

Nota: `b` è assegnato ad `a`, l'espressione `a=b` prende il valore di `b`:

```
if (a=b) => if (b),
True se b != 0, False se b == 0.
```

12.02. Passaggio dell'indirizzo di puntatori

Vedremo meglio di cosa si tratta nel prossimo capitolo, anche se ne abbiamo già accennato in relazione al `scanf()`. È comunque fondamentale ricordare di mettere la "&" nella funzione.

12.03. Mancanza di () per una funzione

Arrivando da precedenti esperienze con altri linguaggi, spesso ci si dimentica di mettere () dopo una funzione; è necessario farlo, anche se non ci sono parametri passati alla funzione stessa.

12.04. Indici di array

È importante ricordare che gli array in C sono dichiarati in maniera diversa: gli array multidimensionali vengono dichiarati alla stessa maniera di quelli semplici, ma con l'elenco dei valori massimi associati ad ogni dimensione; la notazione che viene utilizzata per gli array multidimensionali è la seguente:

```
[ ] [ ] ...
```

dove tra le [] va specificata la dimensione massima per ognuna delle dimensioni dell'array stesso.

Un vettore di "n" elementi ha un intervallo indice che va da 0 a "n-1".

12.05. Array di caratteri e puntatori

Verranno trattati in maniera dettagliata nel prossimo capitolo.

12.06. C è case-sensitive

Una regola fondamentale per l'utilizzo del C è quella di ricordare che le lettere maiuscole e quelle minuscole vengono trattate come fossero caratteri diversi.

12.07. ";" chiude ogni istruzione

È facile dimenticarsene, ma il compilatore risconterà questa mancanza e la segnalerà.

13. Puntatori

I puntatori sono una delle più importanti caratteristiche del C. Se non si è in grado di usare i puntatori in maniera appropriata, non si riusciranno a sfruttare completamente la potenza e la flessibilità che il C permette; infatti, il segreto del linguaggio C sta proprio nel modo in cui utilizza i puntatori.

Il C usa molto i puntatori. Perché?

- è l'unico modo per esprimere alcune operazioni;
- produce codici sorgenti compatti ed efficienti;
- rappresenta uno strumento molto efficace.

Il C utilizza molto i puntatori in maniera esplicita con:

- vettori;
- strutture;
- funzioni.

Nota: i puntatori probabilmente sono la parte del C più difficile da capire; le implementazioni in C sono leggermente diverse rispetto agli altri linguaggi.

13.01. Cos'è un puntatore

Un puntatore è un tipo di dato, una variabile che contiene l'indirizzo in memoria di un'altra variabile. Si possono avere puntatori a qualsiasi tipo di variabile. La dichiarazione di un puntatore include il tipo dell'oggetto a cui il puntatore punta. In C ogni variabile ha due tipi di valori: una locazione e un valore contenuto in quella locazione.

L'operatore & (operatore unario, o monadico) fornisce l'indirizzo di una variabile. L'operatore * (operatore indiretto, o non referenziato) dà il contenuto dell'oggetto a cui punta un puntatore.

Per dichiarare un puntatore ad una variabile, l'istruzione è:

```
int *pointer;
```

Nota: è obbligatorio associare un puntatore ad un tipo particolare; per esempio, non è possibile assegnare l'indirizzo di uno short int ad un long int.

Consideriamo gli effetti del seguente codice:

```
int *pointer; /* dichiara pointer come un puntatore a int */
int x=1,y=2;
```

- (1) pointer= &x; /* assegna a pointer l'indirizzo di x */
- (2) y=*pointer; /* assegna a y il contenuto di pointer */
- (3) x=pointer /* assegna ad x l'indirizzo contenuto in pointer */
- (4) *pointer=3; /* assegna al contenuto di pointer il valore 3 */

Vale la pena considerare cosa succede al "livello macchina" in memoria per capire completamente come funzionano i puntatori. Supponiamo che la variabile x si trovi nella locazione di memoria 100, y nella 200 e pointer nella 1000 (ricordiamo che pointer è una variabile a tutti gli effetti, e così il suo valore necessita di essere memorizzato da qualche parte; è la caratteristica del valore dei puntatori che risulta nuova).

L'istruzione (1) fa sì che pointer punti alla locazione di memoria 100 (quella di x).

La (2) fa sì che y assuma valore 1 (il valore di x).

La (3) fa sì che x assuma valore 100 (cioè il valore di pointer).

La (4) fa sì che il valore del contenuto di pointer sia 3 (quindi x=3).

Notate che le assegnazioni x=1 e y=2 ovviamente caricano questi valori nelle variabili; pointer è dichiarato come puntatore ad un intero e vi è assegnato l'indirizzo di x (&x), così pointer verrà caricato con il valore 100. Successivamente, y prende l'assegnazione del contenuto di pointer. In questo esempio, pointer punta attualmente alla locazione di memoria 100 (la locazione di x). Così ad y viene assegnato il valore di x (chè è 1).

Abbiamo già visto che il C non è molto meticoloso riguardo all'assegnazione di valori di tipo differente. Così è perfettamente legale (sebbene non sia comune a tutti) assegnare il valore corrente di pointer ad x; in questo momento il valore di pointer è 100. Alla fine possiamo assegnare un valore al contenuto di pointer (*ip).

Quindi in merito ai puntatori possiamo avere tre possibili valori:

pointer contenuto o valore della variabile pointer (indirizzo della locazione di memoria a cui punta)

&pointer indirizzo fisico della locazione di memoria del puntatore

*pointer contenuto della locazione di memoria a cui punta

NB. Quando un puntatore viene dichiarato non punta a nulla! Per poterlo utilizzare deve puntare a qualcosa! È infatti un errore comune non assegnare un indirizzo di memoria a un puntatore prima di usarlo. Così:

```
int *ip;
```

```
*ip=100;
```

genererà un errore (crash di programma). L'utilizzo corretto è il seguente:

```
int *ip;
```

```
int x;
```

```
ip=&x;
```

```
*ip=100;
```

Un metodo comune per ovviare al problema dell'assegnazione dell'indirizzo è quello di utilizzare la funzione di libreria standard malloc(), che permette un'allocazione dinamica della memoria; è definita come char *malloc(int number_of_bytes).

Ad esempio:

```
int *p;
p = (int *) malloc(100);
oppure:
p = (int *) malloc(100*sizeof(int))
```

Si possono fare operazioni aritmetiche intere con i puntatori:

```
float *flp, *flq;
*flp=*flp+10;
++*flp;
(*flp)++;
flq=flp;
```

Nota: un puntatore ad una variabile di qualsiasi tipo è un indirizzo in memoria (il quale è un indirizzo intero). Un puntatore per definizione NON è un intero.

La ragione per cui associamo un puntatore ad un tipo di dato è quella per cui è possibile riconoscere quanti bytes contiene il dato. Quando si incrementa un puntatore si cresce il puntatore di un "blocco" di memoria.

Così per un puntatore a char ++ch_ptr aggiunge 1 byte all'indirizzo, per un intero o un float ++ip aggiunge 4 byte all'indirizzo.

Consideriamo una variabile float (fl) ed un puntatore ad un float (flp); ricordiamo che ad un float corrispondono 4 bytes. Assumiamo che flp punti ad fl; se poi incrementiamo il puntatore (++flp), questo si sposta dalla posizione a cui puntava originariamente di 4 bytes in avanti, e punterà quindi al float successivo. D'altra parte, se aggiungiamo 2 al puntatore (flp+2), questo si sposta di due posizioni float, cioè di 8 bytes.

13.02. Puntatori e funzioni

Esamineremo ora la stretta relazione tra i puntatori e le altre parti principali del C, incominciando con le funzioni.

Il C passa argomenti alle funzioni per valore. Ci sono molti casi in cui possiamo avere la necessità di variare un argomento passato in una funzione e ricevere di ritorno il nuovo valore una volta che la funzione è terminata. Gli altri linguaggi sono in grado di fare questa operazione internamente (come ad esempio i parametri "var" in PASCAL), mentre il C utilizza esplicitamente i puntatori per farlo.

Il miglior metodo per comprenderne il funzionamento è quello di fare un esempio in cui dobbiamo essere in grado di ricevere parametri cambiati. Proviamo ad esempio a trovare un modo per effettuare uno scambio di variabili (swap).

La consueta chiamata di funzione:

```
swap(a, b)
```

non funziona. I puntatori forniscono quindi la possibile soluzione: passare l'indirizzo delle variabili alla funzione ed accedere all'indirizzo della funzione stessa. Così la chiamata di funzione nel nostro programma potrà apparire come segue:

```
swap(&a, &b)
```

Il codice sorgente della funzione swap è abbastanza lineare:

```
void swap(int *px, int *py)
{
    int temp;
    temp=*px;      /* contenuto di pointer */
    *px=*py;
    *py=temp;
}

main()
{
```

```

int a=10,b=20;
...
swap(&a,&b);
...
}

```

Possiamo ritornare un puntatore dalle funzioni. Un esempio frequente è quello di ritornare strutture:

```

typedef struct {float x,y,z;} COORD;

main()
{COORD p1, *coord_fn();/*dichiara fn come return pointer di tipo COORD*/
...
p1=*coord_fn(...); /*assegna il contenuto dell'indir. restituito*/
...
}

COORD *coord_fn(...)
{COORD p;
...
p=...; /* assegna un valore alla struttura */
return &p; /* ritorna l'indirizzo di p */
}

```

In questo esempio ritorniamo un puntatore il cui contenuto è immediatamente tradotto in una variabile. Dobbiamo però farlo contestualmente all'uscita del valore dalla funzione, poichè la variabile a cui puntiamo è locale alla funzione stessa che è appena terminata. Questo significa che lo spazio dell'indirizzo si rende subito libero e può essere sovrascritto.

13.03. Puntatori e array

Un'array di elementi può essere pensato come disposto in un insieme di locazioni di memoria consecutive.

Consideriamo il seguente esempio:

```

int a[10],x;
int *ptr;

ptr=&a[0]; /* ptr punta all'indirizzo di a[0] */
x=*ptr; /* x = contenuto di ptr (in questo caso, a[0]) */

```

A questo punto potremo incrementare ptr con successive istruzioni ++ptr ma potremo anche avere (ptr + i) che è equivalente ad a[i], con i=0,1,2,3...9 .

Quindi per raggiungere un elemento qualsiasi dell'array utilizzando un puntatore, l'istruzione può essere:

```
ptr + i = a[i]
```

Attenzione: non c'è alcun limite di controllo per array e pointer, così è facilmente possibile oltrepassare la memoria prevista per un array e sovrascrivere altre cose.

Il C comunque è molto più sottile nei propri collegamenti tra vettori e puntatori. Ad esempio è possibile scrivere:

```
ptr=a;
```

invece di

```
ptr=&a[0];
```

ed a[i] può essere scritto come

```
*(a+i)
```

cioè `&a[i] = a+i`.

Inoltre si possono esprimere puntatori nel seguente modo:

```
ptr[i] = *(ptr+i)
```

Va comunque ricordato che puntatori e vettori sono diversi:

- un puntatore è una variabile, per cui possiamo scrivere:

```
ptr=a ed ptr++
```

- un array non è una variabile quindi:

```
a=ptr ed a++ sono istruzioni non valide
```

Ora siamo in grado di comprendere in che maniera gli array vengono passati alle funzioni. Quando un array è passato ad una funzione, quello che viene effettivamente passato è la locazione in memoria del suo elemento iniziale. Così:

```
strlen(s)=strlen(&s[0])
```

Questo è il motivo per cui dichiariamo la funzione:

```
int strlen(char s[ ] );
```

Una dichiarazione equivalente è:

```
int strlen(char *s);
```

poichè `char s[]=char *s`.

`strlen()` è una funzione della standard library che ritorna la lunghezza di una stringa. Vediamo ora come possiamo scrivere la funzione:

```
int strlen(char *s)
{ char *p=s;

  while (*p != '\0');
  p++;
  return p-s;
}
```

Ora scriviamo una funzione per copiare una stringa in un'altra stringa. `strcpy()` è una funzione della standard library che compie questa operazione:

```
void strcpy(char *s, char *t)
{ while ((*s++ = *t++) != '\0'); }
```

In questo esempio vengono utilizzati puntatori ed assegnazioni per valore. È interessante notare l'utilizzo della frase "null" con `while`.

13.04. Array di puntatori

Visto che i puntatori sono variabili, si possono avere array di puntatori.

Ad esempio:

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

utilizzato per passare argomenti dalla linea di comando.

Gli array di puntatori sono una rappresentazione di dati che può essere convenientemente utilizzata per far fronte in maniera efficiente ai problemi di trattamento di linee di testo con lunghezza variabile (ad esempio, nel caso dell'ordinamento); va ricordato che un testo non può essere spostato o confrontato in una singola operazione.

È possibile risolvere questi problemi con le seguenti operazioni:

- memorizzare le linee end-to-end in un unico array char (\n va utilizzato come separatore delle linee);
- memorizzare i puntatori in un diverso array dove ogni puntatore punta al primo carattere di ogni linea nuova;
- confrontare due linee utilizzando la funzione strcmp() della libreria standard;
- se due linee non sono ordinate, swappare il puntatore nell'array dei puntatori (non in quello del testo).

Questa procedura elimina gli aspetti complicati della gestione della memorizzazione e la dispendiosità dell'operazione di spostamento di linee di testo.

13.05. Array multidimensionali e puntatori

In C dobbiamo pensare agli array multidimensionali in un modo diverso: un array a due dimensioni è un array monodimensionale i cui elementi sono a loro volta degli array. Gli elementi degli array vengono memorizzati riga per riga.

Avevamo visto che per passare un array a una funzione si deve specificare il numero di colonne, mentre non è necessario specificare il numero di righe. La ragione di questo è dovuta ai puntatori, in quanto il C deve sapere il numero di colonne per saltare di riga in riga in memoria.

Si consideri ad esempio di passare l'array a[5][35] ad una funzione f;

si può dichiarare:

```
f(int a[ ][35] ){...}
```

oppure

```
f(int(*a)[35] ){...}
```

Necessitano le parentesi per (*a) perchè il vettore abbia una precedenza maggiore rispetto ad *.

Si noti così la differenza tra:

```
int (*a)[35];   dichiara un puntatore ad un array di 35 int
int *a[35];    dichiara un array di 35 puntatori a int
```

Consideriamo ora la sottile differenza tra puntatori ed array.

Ad esempio:

```
char *name[10];
char Aname[10][20];
```

in C è possibile dichiarare legalmente sia name[3][4] che Aname[3][4].

Comunque:

- "Aname" è un vero array di char a due dimensioni, con 200 elementi;
- l'accesso agli elementi in memoria viene attuato tramite l'istruzione $20 * \text{riga} + \text{colonna} + \text{indirizzo_base}$;
- "name" ha 10 elementi pointer (quindi è un array di puntatori).

Se ogni puntatore nel vettore "name" è settato per puntare ad un array di 20 elementi, solo in quel caso verranno riservati 200 chars (+ 10 elementi).

Il vantaggio di una dichiarazione fatta nel secondo modo è quello che ogni pointer può puntare a vettori di lunghezza diversa.

Un tipico esempio di puntatore ad un array sono le stringhe.

Consideriamo un esempio:

```
#include <stdio.h>
main()
{
    char *s[5];
    *s="ciao";
    printf("%s\n",*s);
}
```

13.06. Inizializzazione statica degli array di puntatori

L'inizializzazione degli array di puntatori è una delle ideali applicazioni per un array interno statico.

Esempio:

```
some_fn()
{
    static char *months = { "no month", "jan", "feb", ... };
}
```

Un array statico riserva un bit di memoria privato permanente.

13.07. Puntatori e strutture

Si tratta di strutture abbastanza lineari e facilmente definibili.

Consideriamo ad esempio:

```
struct COORD {float x,y,z;}pt;
struct COORD *pt_ptr;

pt_ptr=&pt;    /* assegna un puntatore a pt*/
```

L'operatore "->" permette l'accesso a un membro della struttura puntata dal puntatore, cioè:

```
pt_ptr->x=1.0;
pt_ptr->y=pt_ptr->y - 3.1;
```

mentre avevamo visto che l'accesso ai membri di una struttura era dato dall'operatore ".", e cioè:

```
pt.x=2.73;
```

Un esempio può essere costituito dalle Linked Lists:

```
typedef struct { int value;
                ELEMENT *next;
            } ELEMENT;
```

```
ELEMENT n1, n2;
```

```
n1.next = &n2;
```

con cui viene rappresentato il link tra due nodi (n1 ed n2) della struttura ELEMENT; all'interno di ogni nodo di quest'ultima, oltre al valore c'è un puntatore "next" che viene settato all'indirizzo del nodo successivo.

È importante notare che possiamo dichiarare "next" solo come un puntatore ad ELEMENT; non è possibile avere "next" come elemento del tipo della variabile, poichè questo creerebbe una definizione ricorsiva che non è permessa. È invece possibile settare una referenza del pointer poichè vengono messi da parte 4 bytes per ogni puntatore.

Nel prossimo capitolo verrà analizzato ulteriormente questo problema.

13.08. Le "trappole" più comuni dei puntatori

Vogliamo ora puntualizzare due errori solitamente riscontrati nell'utilizzo dei puntatori.

13.08.01. Non assegnare un puntatore ad un indirizzo di memoria prima di utilizzarlo

Un esempio di questo errore:

```
int *x;
*x=100;
```

È necessario però dichiarare una locazione fisica, quindi avremo:

```
int *x;
int y;
x=&y;
*x=100;
```

Può essere difficile individuare questo tipo di errore, poichè nessun compilatore lo segnala. Comunque "x" potrebbe anche avere degli indirizzi random come inizializzazione.

13.08.02. Assegnazione indiretta illegale

Supponiamo di avere una funzione malloc() che prova ad allocare dinamicamente la memoria (in fase di esecuzione) e ritorna un puntatore al blocco di memoria richiesto nel caso in cui termini con successo, oppure un puntatore nullo nell'altro caso.

char *malloc() - una funzione della libreria standard (che vedremo poi)

Abbiamo un puntatore: char *p; consideriamo:

```
*p = (char *) malloc(100); /* richiesta di 100 bytes di memoria */
*p = 'y';
```

C'è un errore in queste istruzioni. Qual è ? Nell'istruzione:

```
*p = (char *) malloc(100);
```

non va messo il carattere "*" associato al puntatore.

Questo è dovuto al fatto che la funzione malloc ritorna un puntatore. Inoltre, "p" non punta a nessun indirizzo. L'istruzione corretta dovrebbe essere:

```
p = (char *) malloc(100);
```

Questa istruzione rivela un ulteriore problema nel caso in cui non ci sia memoria disponibile e "p" sia nullo; perciò non potremo fare l'assegnazione:

```
*p = 'y';
```

Un buon programma C dovrebbe controllare questa possibilità:

```
p = (char *) malloc(100);
if ( p==NULL)
    { printf "Error: Out of Memory \n");
      exit(1);
    }
*p='y';
```

14. Allocazione dinamica della memoria

L'allocazione dinamica è una graziosa e singolare caratteristica del C (rispetto agli altri linguaggi di alto livello). Permette di creare tipi di dati e strutture di qualsiasi dimensione e lunghezza per soddisfare le necessità all'interno dei programmi.

Affronteremo in particolare due applicazioni tra le più diffuse:

- array dinamici;
- strutture dinamiche di dati, cioè linked lists.

14.01. Malloc

La funzione malloc viene comunemente utilizzata soprattutto per "conquistare" una funzione di memoria. Viene definita con l'istruzione:

```
char *malloc(int number_of_bytes)
```

Questa funzione ritorna un puntatore a carattere che corrisponde al punto di inizio in memoria della porzione riservata di dimensione "number_of_bytes". Se la memoria richiesta non può essere allocata, ritorna un puntatore nullo.

Così:

```
char *cp;
cp = malloc(100);
```

tenta di riservare 100 bytes ed assegna l'indirizzo di inizio a "cp".

Se si vuole avere un puntatore ad un altro tipo di dato, si deve utilizzare la coercizione. Inoltre solitamente viene utilizzata la funzione sizeof() per specificare il numero di bytes:

```
int *ip;
ip = (int *) malloc(100*sizeof(int));
```

Il comando (int *) simboleggia la coercizione ad un pointer integer. La coercizione per correggere il tipo dei puntatori è molto importante per garantire che i puntatori aritmetici vengano rappresentati correttamente.

È buona norma utilizzare sizeof anche nel caso in cui si sia già a conoscenza della dimensione reale necessaria; questo garantisce codici portabili (device independent).

"sizeof" può essere usata per trovare la dimensione di un qualsiasi tipo di dato, variabile o struttura; è possibile farlo semplicemente passando uno di questi come argomento alla funzione.

Così:

```
int i;
struct COORD {float x,y,z};
typedef struct COORD PT;
sizeof(int), sizeof(i), sizeof(struct COORD) e sizeof(PT)
```

sono tutti accettabili.

Nell'esempio che segue possiamo utilizzare il collegamento tra pointer e array per trattare la memoria riservata come un array, per poter cioè fare cose come:

```
ip[0] = 100;
```

oppure:

```
for(i=0;i<100;++i) scanf("%d",ip++);
```

14.02. Linked Lists

Riportiamo ora nuovamente un esempio di linked list:

```
typedef struct {int value;
                ELEMENT *next;
            } ELEMENT;
```

Possiamo ora provare a ridefinire la lista dinamicamente:

```
link = (ELEMENT *) malloc(sizeof(ELEMENT));
```

Questo allocherà memoria per un nuovo link.

Se vogliamo togliere la memoria assegnata ad un puntatore, è necessario utilizzare la funzione free():

```
free(link)
```

15. Input ed output

In questo capitolo verranno analizzate le varie strutture di input/output. Abbiamo in precedenza accennato brevemente ad alcune di esse, ma le analizzeremo ora più dettagliatamente.

I programmi C avranno la necessità di includere il file header dello standard I/O, così l'istruzione sarà:

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

15.01. Streams

Le streams costituiscono un mezzo efficace e flessibile per gestire l'I/O (per lettura e scrittura di dati). Una stream è un file o un device fisico (es. printer o monitor) che viene manipolato con un puntatore alla stream. Esiste una struttura di dati interna al C, FILE, che rappresenta tutte le stream ed è definita nel file stdio.h. È sufficiente fare riferimento alla struttura FILE nei programmi C quando si realizza l'I/O utilizzando le stream.

All'interno del programma si deve solamente dichiarare una variabile che punti a tale tipo (non è necessario conoscere alcuna ulteriore specificazione relativa a questa definizione). Si deve aprire una stream prima di eseguire l'I/O, quindi accedervi e poi richiuderla.

Le streams di I/O sono bufferizzate: questo significa che ogni volta viene letto da un file o scritto su di esso un "pezzo" di dimensioni stabilite attraverso alcune aree temporanee di immagazzinamento (è importante notare che il file puntatore punta effettivamente a questo buffer).

Questo metodo rende efficiente l'I/O, ma è necessario fare attenzione: i dati scritti in un buffer non compaiono nel file (o nel device) finchè il buffer non è riempito o scaricato ("n" serve a questo). Qualsiasi uscita anormale del programma può causare problemi.

15.01.01. Streams predefinite

Unix definisce 3 stream predefinite che sono (in `stdio.h`): `stdin`, `stdout`, `stderr` e utilizzano tutte text come metodo di I/O. "`stdin`" e "`stdout`" possono essere usate con files, programmi, device di I/O (come tastiera, console, ...); "`stderr`" va sempre sulla console o sul video.

La console è il default per `stdout` e `stderr`, mentre la keyboard è il default per lo `stdin`. Le streams predefinite vengono aperte automaticamente.

15.01.01.01. Redirezione

Questa è la maniera in cui è possibile variare i default UNIX di I/O. Non si tratta di una parte del C, ma questa operazione dipende dal sistema operativo. Siamo in grado di attuare la redirezione dalla linea di comando:

> - redireziona `stdout` (standard output) in un file. Così, se abbiamo un programma (`out`) che normalmente visualizza sullo schermo, con: `out > file1` l'output verrà inviato in un file (`file1`).

< - redireziona `stdin` (standard input) da un file. Così, se stiamo aspettando un input da tastiera per un programma (`in`), possiamo similmente leggere tale input da un file: `in < file2`.

| - pipe: prende lo `stdout` da un programma e lo trasforma in `stdin` per un altro: `prog1 | prog2`. Se, ad esempio, vogliamo inviare l'output di un programma (solitamente sulla console) direttamente ad una stampante: `out | lpr`

15.02. Funzioni comuni di I/O

Le più comuni funzioni che permettono I/O sono `getchar()` e `putchar()`. Esse sono definite ed usate nel seguente modo:

```
int getchar(void) - legge un char dallo stdin.  
int putchar(char ch) - scrive un char sullo stdout.
```

Ad esempio:

```
int ch;  
  
ch=getchar();  
(void)putchar((char)ch);
```

Funzioni correlate sono:

```
int getc(FILE *stream), int putc(char ch,FILE *stream).
```

La funzione che permette l'output di un elenco di argomenti è:

```
int printf(char *format, arg list ...)
```

che stampa sullo stdout l'elenco di argomenti in accordo al formato specificato. Ritorna il numero di caratteri stampati.

I formati possibili sono:

```
%c per il singolo carattere
%d per numeri decimali
%o per numeri ottali
%x per numeri esadecimali
%u per unsigned int
%f per float o double
%s per stringhe
%e per formato scientifico
```

Tra % e la lettera si può inserire un segno meno che significa giustificazione a sinistra, un numero intero che da l'ampiezza del campo che può essere seguito da un punto e da un altro intero che da il numero di cifre decimali o il numero di caratteri per una stringa.

Ad esempio: `printf("%-3.4f\n",123.987654)` dà come risultato: 123.9876

La funzione che permette l'input di un elenco di variabili è:

```
int scanf(char *format, args....)
```

che legge dallo standard input e assegna all'elenco di variabili i valori letti. Ritorna il numero di caratteri letti.

NB. È richiesto l'indirizzo della variabile o un puntatore.

```
Ad esempio:  int i;
              scanf("%d",&i);
              oppure
              char string[80];
              scanf("%s",string);
```

15.03. Formattazione di I/O

Abbiamo già visto degli esempi di come il C utilizza l'I/O formattato. Ora lo analizzeremo in maniera più dettagliata.

15.03.01. Printf

La funzione è definita come segue:

```
int printf(char *format, arg list ...);
```

e stampa sullo stdout la lista di argomenti conformemente alla stringa di formato specificata. Ritorna il numero di caratteri stampati.

La stringa di formato ha 2 tipi di oggetti:

- caratteri ordinari - questi vengono copiati in output;
- specificazioni di conversione - contraddistinte da "%" e di seguito elencate.

La seguente tabella mostra i possibili formati dei caratteri per le istruzioni printf/scanf:

Formato (%)	Tipo	Risultato
-------------	------	-----------

c	char	singolo carattere
i,d	int	numero decimale
o	int	numero ottale
x,X	int	numero esadecimale (notazione maiuscola o minuscola)
u	int	intero senza segno
s	char *	stampa una stringa terminata con \0
f	double/float	formato -m.ddd...
e,E	"	formato scientifico -1.23e002
g,G	"	"e" o "f" ma più compatti
%	-	stampa il carattere %

Tra il simbolo % ed il carattere di formato, è possibile mettere:

- (segno meno) - giustificazione a sinistra;
- numero intero - ampiezza del campo
- m,d - m=ampiezza del campo, d=precisione del numero di cifre dopo il punto decimale, o numero di caratteri da una stringa

Così, ad esempio, potremo avere:

```
printf("%-2.3f\n",17.23478);
```

e l'output a video sarà:

```
17.235
```

e:

```
printf("VAT=17.5%%\n");
```

dove l'output sarà:

```
VAT=17.5%
```

15.04. Scanf

Questa funzione è definita come segue:

```
int scanf(char *format, args ...)
```

Legge dallo stdin e mette l'input negli indirizzi delle variabili specificate nella lista di args; ritorna il numero di caratteri letti. La stringa di controllo del formato è simile a quella vista per printf.

È importante notare che la funzione scanf richiede di specificare l'indirizzo di ogni variabile, oppure un puntatore ad essa:

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

È anche possibile dare solamente il nome di un array o di una stringa a scanf, poichè questo corrisponde all'indirizzo di partenza dell'array/ stringa:

```
char string[80];
```

```
scanf("%s",string);
```

15.05. Files

I files sono l'esempio più comune di stream. Per aprire un puntatore al file si utilizza la funzione `fopen()` definita come:

```
FILE *fopen(char *name, char *mode)
```

Tale funzione ritorna un puntatore a `FILE`. La stringa "name" è il nome del file su disco a cui vogliamo accedere; la stringa "mode" definisce il tipo di accesso. Se per una qualsiasi ragione il file risulta non accessibile, viene ritornato un puntatore nullo. Le possibili modalità di accesso ai files sono:

- "r" (read),
- "w" (write),
- "a" (append).

Per aprire un file dobbiamo avere una stream (puntatore al file) che punta ad una struttura `FILE`. Così, per aprire in lettura un file chiamato `myfile.dat`, dovremo avere:

```
FILE *stream, *fopen(); /* dichiarazione di una stream e  
                        del prototipo fopen */
```

```
stream = fopen ("myfile.dat", "r");
```

È buona norma controllare l'esito dell'apertura del file:

```
if ((stream = fopen ("myfile.dat", "r"))==NULL)  
    { printf("Can't open %s \n", "myfile.dat");  
      exit(1);  
    }  
...
```

15.05.01. Lettura e scrittura su files

Le funzioni `fprintf` ed `fscanf` sono comunemente utilizzate per l'accesso ai files:

```
int fprintf(FILE *stream, char *format, args ...)  
int fscanf(FILE *stream, char *format, args ...)
```

Sono simili a `printf` e `scanf`, tranne per il fatto che i dati sono letti dalla stream, che deve essere aperta con `fopen()`.

Ad esempio:

```
char *string[80]  
FILE *fp;  
if ((fp=fopen("file.dat", "r")) != NULL)  
    fscanf(fp,"%s",string);
```

Il puntatore alla stream viene incrementato automaticamente con tutte le funzioni di lettura/scrittura su file, quindi non è necessario preoccuparsi di farlo manualmente.

```
char *string[80];  
FILE *stream, *fopen();  
  
if ((stream=fopen(...)) != NULL)
```

```
fscanf(stream,"%s",string);
```

Altre funzioni di I/O da file sono:

```
int getc(FILE *stream),    int fgetc(FILE *stream)
int putc(char ch, FILE *s), int fputc(char ch, FILE *s)
```

Queste funzioni sono come getchar e putchar. "getc" è definita come macro di preprocessore in stdio.h, "fgetc" è una funzione di libreria C; con entrambe si ottiene lo stesso risultato.

Esistono poi le funzioni:

```
fflush(FILE *stream) - per fare la "flush" di una stream
fclose(FILE *stream) - per fare la "close" di una stream
```

Ad esempio:

```
FILE *fp;
if ( (fp=fopen("file.dat","r")) == NULL)
{
    printf("Impossibile aprire file.dat\n");
    exit(1);
}
...
fclose(fp);
```

È possibile accedere alle streams predefinite utilizzando fprintf, etc.:

```
fprintf(stderr,"Cannot Compute!!\n");
fscanf(stdin,"%s",string);
```

15.06. Sprintf ed Sscanf

Simili a fprintf() ed fscanf() sono anche le funzioni:

```
int sprintf(char *string, char *format, args..)
int sscanf(char *string, char *format, args..)
```

che scrivono/leggono su una stringa.

Alcuni esempi:

```
1) int x=10;
   char messaggio[80];
   sprintf(messaggio,"Il valore di x è %d",x);

2) float full_tank = 47.0; /* litri */
   float miles = 300;
   char miles_per_litre[80];
   sprintf(miles_per_litre,"Miles per litre = %2.3f", miles/full_tank);
```

15.07. Input dalla linea di comando

Il C permette di leggere argomenti dalla linea di comando, e questi possono poi essere utilizzati all'interno dei programmi. In fase di lancio del programma, possiamo scrivere gli argomenti dopo il nome del programma da eseguire. Abbiamo visto un esempio di questa possibilità in relazione all'utilizzo dei compilatori:

```
c89 -o prog prog.c
```

dove "c89" è il programma, mentre "-o prog prog.c" sono gli argomenti. Al fine di essere in grado di utilizzare tali argomenti, è necessario definirli nel seguente modo:

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

così la funzione main ha ora i propri argomenti; questi sono gli unici argomenti main accettati.

In questa definizione:

- argc è il numero degli argomenti digitati, incluso il nome del programma;
- argv è un array di stringhe contenente ciascun argomento, compreso il nome del programma come primo elemento.

Ad esempio:

```
#include <stdio.h>
main(int argc, char **argv)
{ /* programma per stampare gli argomenti dalla linea di comando */
  int i;
  printf("argc=%d\n",argc);
  for(i=0;i < argc;++i)
    printf("argv[%d]:=%s\n",i,argv[i]);
}
```

Se si è compilato, chiamandolo args e fatto eseguire scrivendo:

```
args f1 "f2" f3 4 stop!
```

l'output sarà:

```
argc=6
argv[0]=args
argv[1]=f1
argv[2]=f2
argv[3]=f3
argv[4]=4
argv[5]=stop!
```

Va notato che:

- argv[0] è il nome del programma;
- argc totalizza anche il nome del programma;
- tra gli argomenti, i caratteri "" vengono ignorati (sono considerati solamente delimitatori di argomenti);
- gli spazi bianchi delimitano gli argomenti;
- nel caso in cui sia necessario mantenere spazi bianchi, occorre metterli tra "".

15.08. I/O di basso livello

Tale forma di I/O è UNBUFFERED, cioè ogni richiesta di read/write comporta un accesso diretto al disco (o device) scrivendo o leggendo uno specificato numero di bytes. Non ci sono facilitazioni di formato, poiché a questo livello si lavora con i bytes di informazione; questo significa che ora si usano binary (e non text) files. Invece di un puntatore a

file si usa un trattamento del file di basso livello, detto anche "descrittore del file" che dà un unico numero intero per identificare ciascun file. Per aprire un file si usa:

```
int open(char *filename, int flag, int perms)
```

che ritorna un file descriptor, oppure -1 se l'operazione fallisce.

Il flag controlla l'accesso al file ed ha i seguenti predefiniti valori definiti nel file `fcntl.h`:

```
O_APPEND, O_CREAT, O_EXCL, O_RDONLY, O_RDWR, O_WRONLY ecc.
```

"perms" viene settato ottimamente a 0 per la maggior parte delle applicazioni.

Per creare un file si può usare la funzione:

```
creat(char *filename, int perms)
```

Per chiudere un file si usa:

```
int close(int handle)
```

Per leggere/scrivere uno specificato numero di bytes da/su un file immagazzinati in una locazione di memoria specificata da "buffer" si utilizzano:

```
int read(int handle, char *buffer, unsigned length)
```

```
int write(int handle, char *buffer, unsigned length)
```

Queste due funzioni ritornano il numero di byte letti/scritti o -1 se falliscono.

Per specificare la lunghezza si utilizza, in genere, la funzione `sizeof()`.

Ad esempio:

```
/* legge un elenco di float da un file binario il primo byte del file dice quanti float ci sono  
nel file. Successivamente vengono elencati i float; il nome del file è letto dalla linea di comandi */
```

```
#include <stdio.h>
#include <fcntl.h>
float bigbuff[1000];

main(int argc, char **argv)
{
    int fd;
    int bytes_read;
    int file_length;

    if((fd=open(argv[1], O_RDONLY))===-1)
    { /* errore, file non aperto */
        exit(1);
    }
    if ((bytes_read=read(fd, &file_length, sizeof(int))==-1)
    { /* errore in lettura file */
        exit(1);
    }
    if (file_length>999)
    { /* file troppo grande */
        exit(1);
    }
}
```

```

    }
    if((bytes_read=read(fd,bigbuff,
        file_length*sizeof(float))!=-1)
    { /* errore in lettura file aperto */
        exit(1);
    }
}

```

16. Il preprocessore C

La chiamata al preprocessore è il primo passo da compiere fra i passi per la compilazione di un programma C (si tratta di una caratteristica presente solo nei compilatori C). Il preprocessore fornisce un proprio linguaggio, il quale può costituire un potente strumento per i programmatori. Ricordiamo che tutte le istruzioni e i comandi del preprocessore cominciano con un #.

L'utilizzo del preprocessore è vantaggioso, poichè rende:

- i programmi più facili da sviluppare,
- più facili da leggere,
- più facili da modificare,
- il codice C più trasportabile tra le diverse architetture macchina.

Il preprocessore permette anche di "customizzare" il linguaggio. Ad esempio, per sostituire {...} blocchi di istruzioni delimitati con la notazione Pascal (come begin ... end), è sufficiente dichiarare:

```

#define begin {
#define end }

```

Durante la compilazione tutte le occorrenze di begin/end vengono sostituite con i corrispondenti { o }; così la successiva fase di compilazione C non riconoscerà alcuna differenza di linguaggio.

16.01. #define

Viene utilizzato per definire costanti, oppure qualsiasi sostituzione macro. Va utilizzata come segue:

```

#define <macro> <nome-sostituzione>

```

Ad esempio:

```

#define FALSE 0
#define TRUE !FALSE

```

È possibile anche definire delle piccole funzioni utilizzando l'istruzione #define. Se, ad esempio, vogliamo trovare il massimo tra due variabili:

```

#define max(A,B) ((A)>(B) ? (A):(B))

```

(ricordiamo che "?" in C corrisponde all'operatore ternario).

Questa istruzione, però, non definisce propriamente una funzione "max"; significa invece che in qualsiasi posto noi richiamiamo `max(var1,var2)`, il testo viene sostituito dalla definizione appropriata (`var1` e `var2` non devono necessariamente essere i nomi delle variabili). Così se nel nostro codice C scriviamo ad esempio:

```
x=max(q+r,s+t);
```

dopo la chiamata al preprocessore, se fossimo in grado di vedere il codice, questo apparirebbe nel seguente modo:

```
x=( (q+r) > (r+s) ? (q+r) : (s+t) );
```

16.02. #undef

Questo comando esegue l'undefine di una macro; per poterla ridefinire ad un differente valore, una macro deve essere undefined.

16.03. #include

Questo comando include un file all'interno del codice. Ci sono due possibili forme:

```
#include <file>
```

oppure

```
#include "file"
```

`<file>` indica al compilatore di cercare dove sono memorizzati i files di include di sistema; solitamente i sistemi UNIX memorizzano i files nella directory `/usr/include`; "file" cerca un file nella directory corrente (quella da cui il programma viene eseguito).

I files inclusi di solito contengono prototipi C e dichiarazioni da file header e non codici C algoritmici.

16.04. #if - Inclusione condizionale

`#if` valuta una costante espressione intera; è necessario utilizzare `#endif` per delimitare la fine dell'istruzione. È possibile anche avere `else` (con `#else`) ed `else if` (con `#elif`). Altro uso comune che può essere fatto con `#if` è il seguente:

```
#ifdef      - if defined
#ifndef     - if not defined
```

Queste istruzioni sono utili per controllare se le macro sono settate, magari da differenti moduli di programma e da file header.

Ad esempio:

```
#ifdef USESTRINGDOTH
#include <string.h>
#else USESTRINGDOTH
#include <strings.h>
```

```
#endif USESTRINGDOTH
```

Ad esempio, per settare la dimensione degli integer per un programma C portabile tra TurboC (su MS-DOS) e il sistema operativo Unix (o altro); ricordiamo che TurboC usa gli interi a 16 bit mentre Unix utilizza gli interi a 32 bit. Presumiamo che se TurboC sta girando, una macro "TURBOC" risulterà definita; così dobbiamo solamente preoccuparci di controllare questo:

```
#ifdef TURBOC
#define INT_SIZE 16
#else
#define INT_SIZE 32
#endif
```

Come ulteriore esempio, potremmo avere la necessità di includere il file `msdos.h` in sostituzione del file `default.h` nel caso in cui si stia eseguendo il programma su una macchina MS-DOS. Una macro "SYSTEM" è settata al tipo di sistema, così è sufficiente controllare:

```
#if SYSTEM == MSDOS
#include <msdos.h>
#else
#include "default.h"
```

17. Scrittura di grossi programmi

In questo capitolo verranno trattati gli aspetti teorici e pratici che devono essere considerati quando si scrivono grossi programmi. In questi casi è consigliabile suddividere i programmi in moduli, che dovrebbero essere in file sorgenti separati. L'istruzione `main()` sarà in un solo file (rappresenta `main.c`), mentre tutti gli altri conterranno delle funzioni.

È possibile creare una propria libreria di funzioni scrivendo un gruppo di subroutine in uno o più moduli. Infatti i moduli possono essere condivisi da diversi programmi semplicemente includendoli in fase di compilazione, come vedremo. Ci sono molti vantaggi legati a questo modo di operare:

- i moduli verranno naturalmente divisi in gruppi comuni di funzioni;
- è possibile compilare ogni modulo separatamente e linkarlo poi nei moduli compilati (come vedremo più avanti);
- le utility UNIX, come `make`, aiutano a mantenere grossi sistemi (anche questo verrà analizzato all'interno di questo capitolo).

17.01. File header

Se adottiamo un approccio modulare, allora risulterà spontaneo mantenere all'interno di ogni modulo la definizione delle variabili, i prototipi di funzioni, e così via. Comunque sorge un problema nel caso in cui più moduli necessitino la condivisione di tali definizioni. È consigliabile centralizzare la definizione in un file e condividerlo poi con gli altri moduli. I file di questo tipo sono chiamati solitamente "header file". Le convenzioni stabiliscono che questi file abbiano un suffisso ".h".

Abbiamo già incontrato in precedenza file header delle librerie standard, come ad esempio:

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

Siamo in grado di definire dei file header propri, ed includerli poi nei programmi con un'istruzione del tipo:

```
#include "my_head.h"
```

È importante notare che il file header solitamente contengono solo definizioni di tipi di dati, prototipi di funzioni e comandi per il preprocessore C. Se abbiamo, ad esempio, tre moduli:

```
main.c
WriteMyString.c
header.h
```

solitamente ogni singolo modulo verrà compilato separatamente. Alcuni moduli hanno un `#include "header.h"` per accedere alle definizioni comuni. Alcuni altri, come `main.c`, includono anche file header standard. Nell'esempio accennato, "main" richiama la funzione `WriteMyString.c()` che è nel modulo `WriteMyString.c`. In quest'ultima funzione potrebbe essere richiamato un prototipo di funzione "void" che viene definito in `header.h`. Notiamo che in generale è necessario decidere tra il desiderio che ogni modulo ".c" possa accedere alle informazioni di cui necessita unicamente per il proprio lavoro, e la realtà pratica di mantenere molti file header. Per i programmi di moderate dimensioni, probabilmente è meglio mantenere uno o due file header che condividano le definizioni di più di un modulo.

Un problema finora tralasciato in merito all'approccio modulare riguarda le variabili di sharing. Se abbiamo delle variabili globali dichiarate ed utilizzate nel modulo corrente, in che modo è possibile fare riconoscere tali variabili agli altri moduli? Possiamo passare i valori come parametri delle funzioni, ma:

- questa tecnica può risultare molto laboriosa se passiamo gli stessi parametri a molte funzioni e/o se sono coinvolti elenchi di argomenti piuttosto lunghi;

- è difficile memorizzare localmente vettori molto grandi oppure strutture (ci sono problemi di memoria con le pile).

17.02. Variabili esterne e funzioni

L'aggettivo "interno" implica che gli argomenti e le funzioni vengano definite all'interno delle funzioni stesse (local). Le variabili "esterne" sono definite al di fuori della funzione; queste sono potenzialmente disponibili per l'intero programma (global), ma non necessariamente lo sono. Le variabili esterne sono sempre fisse. Sottolineiamo il fatto che in C (a differenza del Pascal) tutte le definizioni di funzioni sono esterne.

17.02.01. Scopo delle variabili esterne

Una variabile esterna (o funzione) non è sempre completamente globale. Il C applica la seguente regola: l'estensione di una variabile (o di una funzione) esterna comincia dal suo punto di dichiarazione e termina alla fine del file (modulo) in cui viene dichiarata. Consideriamo il seguente esempio:

```
main()
{ ... }

int what_scope;
float end_of_scope[10];

void what_global()
{ ... }

char alone;

float fn()
{ ... }
```

`main` non può vedere `what_scope` o `end_of_scope`, mentre le funzioni `what_global` ed `fn` possono vederle. Solo la funzione `fn` può vedere la variabile `alone`.

Questa è anche la prima delle ragioni per cui dobbiamo creare prototipi di funzioni prima che nel codice venga dichiarato il corpo della funzione.

In questo caso `main` non riconoscerà le funzioni `what_global` ed `fn`; a sua volta `what_global` non riconosce `fn`, ma `fn` riconosce invece `what_global`, poichè è stata dichiarata al di sopra di essa.

Facciamo presente che l'altra ragione per cui creiamo prototipi di funzioni è che possono essere fatti alcuni controlli sui parametri passati alle funzioni.

Se abbiamo bisogno di riferirci ad una variabile esterna prima della sua dichiarazione oppure nel caso in cui sia definita in un altro modulo, dobbiamo dichiararla come una variabile esterna, cioè:

```
extern int what_global;
```

In questo modo ritorniamo all'esempio modulare. Abbiamo una stringa global AnotherString dichiarata in main.c e condivisa con WriteMyString.c, dove è dichiarata come variabile esterna.

Attenzione: il prefisso "extern" è una dichiarazione e non una definizione, cioè nessun blocco di memoria viene riservato per una variabile esterna (si tratta solamente della dichiarazione della proprietà della variabile).

La variabile vera e propria deve essere definita una sola volta all'interno dell'intero programma, mentre è possibile avere tutte le dichiarazioni esterne che sono necessarie.

Le dimensioni degli array devono ovviamente essere date con le definizioni, ma non sono richieste con le dichiarazioni esterne.

Ad esempio:

```
main.c: int arr[100];  
file.c: extern int arr[ ];
```

17.03. L'utility Make

L'utility Make è un intelligente program manager che mantiene l'integrità di un gruppo di moduli di programma, una raccolta di programmi oppure un sistema completo. Nella pratica non è possibile avere programmi che appartengano ad un sistema di file qualsiasi (ad esempio, i capitoli di testo in un libro, già passati in composizione).

Questa utility viene principalmente utilizzata come aiuto in fase di sviluppo di sistemi. Make è stata originariamente sviluppata per UNIX, ma attualmente è disponibile sulla maggior parte dei sistemi.

Facciamo presente che make è un'utility per la programmazione, e non una parte del linguaggio C oppure un qualsiasi linguaggio per la soluzione di un determinato problema.

Consideriamo il problema di mantenere un grosso numero di file sorgenti:

```
main.c f1.c ... fn.c
```

per cui potremmo normalmente compilare sul nostro sistema con il comando:

```
cc -o main main.c f1.c ... fn.c
```

Comunque, se siamo a conoscenza del fatto che alcuni file sono già stati precedentemente compilati ed i loro sorgenti non sono variati da quando possiamo aver provato, e vogliamo salvare la compilazione totale linkando questi file nel codice oggetto, il comando sarà il seguente:

```
cc -o main main.c f1.c ... fi.o ... fj.o ... fn.c
```

Possiamo usare l'opzione "-c" del compilatore C per creare un file ".o" per un modulo stabilito. Ad esempio:

```
cc -c main.c
```

creerà un file main.o. In questo caso non c'è la necessità di fornire nessun link alle librerie, poichè questo problema viene automaticamente risolto nella fase di link della compilazione.

Comunque, esiste un problema per la compilazione dell'intero programma in questo modo piuttosto lungo:

- è tempo sprecato compilare un modulo .c: se il modulo è già stato compilato in precedenza e non è stato successivamente variato, non c'è alcuna necessità di ricompilarlo. È sufficiente linkare i file oggetto. Potrebbe comunque non essere facile ricordare quali file sono realmente aggiornati; se li linkiamo in un vecchio file oggetto, il nostro programma eseguibile finale potrebbe risultare errato;

- è piuttosto laborioso (ed incline agli errori) digitare una lunga sequenza di compilazione sulla linea di comando. Molti dei nostri file potrebbero dover essere linkati, così pure come molti file delle librerie di sistema: può così risultare molto difficile ricordare la corretta sequenza delle operazioni. Anche nel caso in cui si faccia un cambiamento

minimo alla linea di comando utilizzando l'editing di sistema, si possono facilmente fare errori.

Se usiamo l'utility make, questa provvede automaticamente a fare tutti questi controlli. In generale, questa utility garantisce la ricompilazione solamente dei moduli che hanno i file oggetto più vecchi rispetto ai file sorgenti.

17.04. Programmazione di Make

La programmazione di make è abbastanza lineare; fondamentalmente, dobbiamo scrivere una sequenza di comandi che descrivano come il nostro programma (o un sistema di programmi) può essere costruito a partire dai file sorgenti.

La sequenza di costruzione viene descritta nel file "makefile", che contiene regole di dipendenza e regole di interpretazione.

Una regola di dipendenza ha due parti (una parte destra ed una sinistra, separate da ":"):

```
left_side : right_side
```

La parte sinistra è formata dal nome di un target (nome del programma o del file di sistema) che deve essere creato, mentre la parte destra fornisce i nomi dei file da cui dipende il file di target (ad esempio, file sorgenti, file header o file di dati).

Se il file target risulta non aggiornato rispetto alle parti che lo costituiscono, è necessario sottostare alle regole di interpretazione (o di costruzione) che seguono le regole di dipendenza.

In questo modo, nel caso di un tipico programma C, quando si esegue il makefile vengono seguiti questi passi:

- Viene letto il makefile: questo riporta quali oggetti e file di libreria necessitano di essere linkati e quali file header e sorgenti devono essere compilati per creare ogni file oggetto;

- Data e ora di ogni file oggetto vengono controllati con quelli di ogni file sorgente e file header da cui dipende. Se un qualsiasi file sorgente o header risulta più recente rispetto al file oggetto, allora i file sono stati modificati dopo l'ultima compilazione e perciò viene ricompilato il file oggetto;

- Una volta che tutti i file oggetto sono stati controllati, vengono controllati data e ora di tutti i file oggetto e confrontati con quelli del file eseguibile. Se uno qualsiasi dei file risulta più recente, allora i file oggetto verranno ricompilati.

Facciamo presente che i file di make possono obbedire ad un qualsiasi comando che venga digitato sulla linea di comando. Perciò possiamo usare i makefile per fare molto più che compilare un modulo sorgente del sistema. Per esempio, potremmo fare backup di file, eseguire programmi se i file di dati risultano variati, oppure ripulire directory.

17.05. Creazione di un makefile

Questa operazione risulta piuttosto semplice: si tratta di creare un file di testo utilizzando un qualsiasi text editor. Il makefile contiene solamente un elenco di file collegati ed i comandi necessari per soddisfare tali collegamenti.

Riportiamo ora un esempio di makefile:

```
prog.o f1.o f2.o
c89 prog.o f1.o f2.o -lm etc.
```

```
prog.o: header.h prog.c
c89 -c prog.c
```

```
f1.o: header.h f1.c
c89 -c f1.c
```

```
f2.o: ...
...
```

Make interpreterà il file nel seguente modo:

- prog dipende da 3 file: prog.o, f1.o ed f2.o. Se uno qualsiasi dei file oggetto sono stati modificati dopo l'ultima compilazione, i file devono essere linkati nuovamente;

- prog.o dipende da 2 file. Se questi risultano modificati, prog.o deve essere ricompilato; la stessa cosa vale per f1.o ed f2.o.

Gli ultimi 3 comandi nel makefile sono chiamati "regole esplicite", poichè i file nei comandi vengono elencati con il proprio nome.

Possiamo anche usare "regole implicite", le quali ci permettono di generalizzare le nostre regole e memorizzare ciò che è stato digitato.

È possibile prendere:

```
f1.o: f1.c
cc -c f1.c
```

```
f2.o: f2.c
cc -c f2.c
```

e generalizzarlo con il seguente comando:

```
.c.o: cc -c $<
```

Questo si legge come ".source_extension.target_extension: command". "\$<" è l'abbreviazione per il nome file con estensione ".c".

È possibile aggiungere commenti in un makefile utilizzando il simbolo "#"; in questo modo, tutti i caratteri che seguono # sulla linea vengono ignorati.

Make ha molti comandi interni simili o addirittura uguali a quelli UNIX. Alcuni esempi:

```
break, date, mkdir, type, chdir, mv (move o rename),
cd, rm (remove), cp (copy), path
```

17.06. Macro di Make

Utilizzando make è possibile definire delle macro; queste vengono usate solitamente per memorizzare i nomi dei file sorgenti, i nomi dei file oggetto, le opzioni del compilatore ed i link alle librerie.

Le macro sono semplici da definire; ad esempio:

```
SOURCES = main.c f1.c f2.c
CFLAGS = -g -C
LIBS = -lm
PROGRAM = main
OBJECTS = (SOURCES: .c=.o)
```

dove (SOURCES: .c=.o) trasforma le estensioni ".c" dei file elencati in SOURCES in estensioni ".o".

Per referenziare o richiamare una macro in make, è necessario \$(macro_name); ad esempio:

```
$(PROGRAM) : $(OBJECTS)
$(LINK.C) -o $@ $(OBJECTS) $(LIBS)
```

È importante notare:

- \$(PROGRAM) : \$(OBJECTS) - crea un elenco di dipendenze ed oggetti;
- l'utilizzo di una macro interna, cioè \$.

Ci sono molte macro interne; alcune delle più comuni sono:

- \$star - parte del file name nella directory corrente (meno .suffisso)
- \$@ - nome completo dell'oggetto corrente
- \$< - file .c dell'oggetto

17.07. Esecuzione di Make

È sufficiente digitare make dalla linea di comando. UNIX cerca automaticamente un file di nome "Makefile" (notare l'iniziale maiuscola, mentre tutto il resto è minuscolo). Il Makefile presente nella directory corrente verrà eseguito.

È possibile annullare questa ricerca di un file semplicemente digitando il comando "make -f make_filename". Ad esempio:

```
make -f my_make
```

Per quanto riguarda i makefile, esistono altre opzioni (-option) oltre a quella appena vista.

18. UNIX e il C

C'è un collegamento molto stretto tra il C e la maggioranza dei sistemi operativi che eseguono i nostri programmi in C. In questo capitolo verranno analizzate le modalità con cui il C ed UNIX interagiscono. Dobbiamo utilizzare UNIX per mantenere il nostro spazio per i file, per editare, compilare ed eseguire programmi, e così via. Ma UNIX porta molti più vantaggi che questi.

18.01. Vantaggi di usare UNIX con il C

- Portabilità:

UNIX, o una delle varietà di UNIX, è disponibile su molte macchine. I programmi scritti in standard UNIX e C possono essere eseguiti su una qualsiasi macchina con minima difficoltà.

- Multiuser/Multitasking:

Molti programmi sono in grado di utilizzare le elevate capacità di elaborazione delle macchine.

- Trattamento dei File:

File system gerarchico, con molte routine per il trattamento dei file.

- Programmazione Shell:

UNIX fornisce un potente interprete di comandi che comprende oltre 200 comandi e può anche eseguire programmi sia UNIX che definiti dall'utente.

- Pipe:

L'output di un programma può essere utilizzato come input per un altro programma. Questo può essere fatto dalla linea di comando, oppure all'interno di un programma C.

- Utility UNIX:

Ci sono oltre 200 utility che permettono di realizzare molte routine senza scrivere dei nuovi programmi (come, ad esempio: make, grep, diff, awk, more, ...)

- Chiamate di sistema:

UNIX ha circa 60 chiamate di sistema residenti nel "cuore" del sistema operativo, o kernel di UNIX. Le chiamate sono scritte in C, e sono accessibili dai programmi C. Esempi di queste chiamate possono essere gli I/O di base e il

clock di sistema. La funzione open() è un esempio di una chiamata di sistema.

- Funzioni di libreria:

Aggiunte al sistema operativo.

18.02. Utilizzo delle chiamate di sistema UNIX e delle funzioni di libreria

Per utilizzare le chiamate di sistema e le funzioni di libreria in un programma C è sufficiente richiamare la funzione C appropriata. Abbiamo già visto alcune chiamate di sistema trattando l'I/O di basso livello (open(), creat(), read(), write() e close() sono esempi).

Abbiamo invece incontrato esempi di funzioni delle librerie standard nella parte dedicata alle funzioni di I/O di alto livello (fopen(), fprintf(), sprintf(), malloc(), ...).

Tutte le funzioni matematiche (come sin(), cos(), sqrt()) ed i generatori di numeri random (random(), seed(), lrand48(), drand48(), ...) sono funzioni della libreria standard math.

È da notare il fatto che la maggior parte delle funzioni delle librerie standard utilizzeranno delle chiamate di sistema all'interno di esse. Per molte chiamate di sistema e funzioni di libreria, è necessario includere un file header appropriato, come ad esempio stdio.h o math.h.

Informazioni su quasi tutte le chiamate di sistema e le funzioni di libreria sono disponibili sulle pagine del manuale. Queste sono disponibili on-line; è sufficiente digitare il nome della funzione "man".

Ad esempio:

```
man drand48
```

darà informazioni in merito a questo generatore di numeri random.

Tutte le chiamate di sistema e le funzioni di libreria verranno elencate in seguito. Abbiamo già visto esempi di funzioni di libreria per il trattamento delle stringhe; più avanti vedremo l'applicazione di alcune altre funzioni di libreria e di sistema.

18.03. Trattamento di file e directory

Ci sono molte utility UNIX che permettono la gestione di directory e file. cd, ls, rm, cp, mkdir, etc. sono esempi solitamente molto noti. Vedremo ora come sia possibile raggiungere un simile scopo dall'interno di un programma C.

18.03.01. Funzioni di trattamento delle directory

Queste operazioni coinvolgono fondamentalmente chiamate ad appropriate funzioni.

Ad esempio:

```
int chdir(char *path) - cambia la directory a quella specificata nella
stringa "path"
```

Esempio: emulazione C del comando UNIX "cd"

```
#include <stdio.h>

main(int argc, char **argv)
{
    if (argc < 2)
        {printf("Usage: %s <pathname> \n", argv[0]);
        exit(1);
        }
    if (chdir(argv[1]) != 0)
        {printf("Error in chdir\n");
        exit(1);
        }
```

```

    }
}

```

```
char *getwd(char *path)
```

Restituisce il pathname completo della directory di lavoro corrente. "path" è un puntatore ad una stringa in cui viene ritornato il pathname. "getwd" ritorna un puntatore alla stringa, oppure NULL nel caso si verifichi un errore.

```
scandir(char *dirname, struct direct **namelist, int (*select)(), int (*compar)())
```

Legge il dirname della directory e crea un array di puntatori ad ogni voce contenuta della directory, oppure "-1" per un errore. "namelist" è un puntatore ad un array di puntatori alla struttura. (*select)() è un puntatore ad una funzione che viene richiamata con un puntatore ad ogni voce della directory (definita in <sys/types>) e potrebbe ritornare un valore diverso da 0 se la voce della directory è inclusa nell'array. Se il puntatore è nullo, allora verranno incluse tutte le voci della directory. L'ultimo argomento è un puntatore ad una routine che viene passato a qsort (vedere "man qsort") - una funzione interna che sorta l'array completo. Se il puntatore è NULL, l'array non viene sortato.

```
alphasort(struct direct **d1, **d2)
```

"alphasort()" è una routine interna che sorta alfabeticamente un array.

Esempio: una semplice versione dell'utility UNIX "ls"

```

#include <sys/types.h>
#include <sys/dir.h>
#include <sys/param.h>
#include <stdio.h>

#define FALSE 0
#define TRUE !FALSE

extern int alphasort();

char PATHNAME[MAXPATHLEN];

main() {int count,i;
        struct direct **files;
        int file_select();

        if(getwd(pathname) == NULL)
            {printf("Error getting path\n");
             exit(0);
            }
        printf("Current Working Directory=%s\n",pathname);
        count=scandir(pathname,&files,file_select,alphasort);
            /* if no files found, make a non-selectable menu item */
            if (count <= 0)
                {printf("No files in this directory\n");
                 exit(0);
                }
            printf("Number of files=%d\n",count);
            for (i=1;i<count+1;++i)
                printf("%s",files[i-1]->d_name);
            printf("\n"); /* flush buffer */
        }

int file_select(struct direct *entry)

    {if ((strcmp(entry->d_name, ".") == 0) ---
        (strcmp(entry->d_name, "..") == 0))
        return (FALSE);
    else

```

```

    return (TRUE);
}

```

"scandir" ritorna la directory corrente (.) e la directory di livello superiore (..), come pure tutti i files contenuti; così è necessario controllare l'output, e ritornare FALSE in modo che le due directory non vengano incluse nella lista di file.

Facciamo presente che "scandir" e "alphasort" hanno le definizioni nei files sys/types.h e sys/dir.h. MAXPATHLEN e "getwd" sono definite in sys/param.h.

Possiamo avanzare ulteriormente e cercare files specifici. Modificheremo ora file_select() in modo che scandisca solamente i file con suffisso .c, .o o .h:

```

int file_select(struct direct *entry)

{char *ptr;
char *rindex(char *s, char c);

if ((strcmp(entry->d_name, ".") == 0) ---
    (strcmp(entry->d_name, "..") == 0))
    return(FALSE);

/* Check for filename extensions */
ptr=rindex(entry->d_name, '.')
if ((ptr != NULL) &&
    ((strcmp(ptr, ".c") == 0)
    || (strcmp(ptr, ".h") == 0)
    || (strcmp(ptr, ".o") == 0) ))
    return(TRUE);
else
    return(FALSE);
}

```

Facciamo notare che rindex() è una funzione di trattamento delle stringhe, che ritorna un puntatore all'ultima occorrenza del carattere "c" nella stringa "s", oppure un puntatore NULL se "c" non è presente nella stringa (index() è una funzione simile, ma assegna ad un puntatore la prima occorrenza).

18.03.02. Routine di trattamento dei file

Esiste un modo per determinare le possibilità di accesso ai file:

```
int access(char *path, int mode)
```

"path" punta ad un path-name che individua il file, "access()" controlla il file nominato per verificarne l'accessibilità conformemente a "mode", definito in "#include <unistd.h>":

```

R_OK   - test per il permesso di lettura;
W_OK   - test per il permesso di scrittura;
X_OK   - test per il permesso di esecuzione o ricerca;
F_OK   - test sia per verificare che le directory di livello superiore al file abbiano il permesso di ricerca, e
        che il file esista.

```

"access()" ritorna: 0 in caso di successo, -1 in caso di insuccesso e setta "errno" per indicare l'errore. Per l'elenco degli errori, vedere le pagine di "man".

18.03.03. errno

Si tratta di una speciale variabile di sistema che viene settata nel caso in cui una chiamata di sistema non possa eseguire l'insieme dei propri compiti. Per utilizzare "errno" in un programma C, deve essere dichiarato con l'istruzione:

```
extern int errno;
```

Può essere manualmente azzerato all'interno di un programma C, altrimenti questo ritorna semplicemente il proprio ultimo valore.

```
int chmod(char *path, int mode)
```

cambia il modo di accesso di un file, specificato da "path", al "mode" stabilito.

chmod() ritorna 0 in caso di successo, -1 nel caso contrario e setta "errno" ad indicare l'errore riscontrato. Gli errori sono definiti in "#include <sys/stat.h>".

Il modo di accesso di un file può essere settato utilizzando macro predefinite in sys/stat.h (vedere le pagine "man"), oppure settando la modalità in un numero ottale di 3 cifre. La cifra più a sinistra specifica i privilegi del proprietario, quella centrale i privilegi del gruppo e quella più a destra i privilegi degli altri utenti. Per ogni cifra ottale intendiamo un numero binario di 3 bit. Il bit più a sinistra corrisponde all'accesso per la lettura (on/off), quello centrale alla scrittura, quello a destra all'esecuzione.

Così:

```
4 (ottale 100) = sola lettura
2 (010)      = scrittura
6 (110)      = lettura e scrittura
1 (001)      = esecuzione
```

Con modalità di accesso 600 daremo i permessi di lettura e scrittura al proprietario, mentre gli altri non avranno nessun permesso. 666 dà a tutti l'accesso in lettura/scrittura.

Ricordiamo che esiste anche un comando UNIX "chmod".

```
int stat(char *path, struct stat *buf),
int fstat(int fd, struct stat *buf)
```

Con "stat()" si ottengono informazioni in merito al file nominato con "path". Il permesso di lettura, scrittura o esecuzione del file specificato non è richiesto, ma tutte le directories elencate nel path-name per arrivare al file devono avere il permesso di ricerca.

Con "fstat()" si ottengono le stesse informazioni riguardo ad un file aperto indicato con la descrizione dell'argomento della funzione, allo stesso modo di come potrebbero essere ottenute con una chiamata "open" (I/O di basso livello).

"buf" è un puntatore ad una struttura "stat" in cui vengono memorizzate le informazioni relative al file.

Una struttura stat è definita in "#include <sys/types.h>" (vedere le pagine "man" per ulteriori informazioni).

"stat()" e "fstat()" ritornano 0 in caso di successo, -1 in caso di insuccesso e settano "errno" per indicare l'errore verificatosi. Anche gli errori sono definiti in "#include <sys/stat.h>".

```
int unlink(char *path)
```

elimina il link esistente relativo alla directory indicata con "path".

"unlink()" ritorna 0 in caso di successo, -1 in caso di insuccesso e setta "errno" per indicare l'errore. Gli errori sono elencati in "#include <sys/stat.h>".

18.04. Controllo e gestione dei processi

Un processo può essere definito fondamentalmente come "ogni singolo programma in esecuzione". Può essere un programma di sistema (come login, update e csh), oppure programmi lanciati dall'utente (textedit, dbxtool o un programma scritto dall'utente stesso).

Quando UNIX esegue un processo, assegna ad ognuno un numero unico ed univoco, cioè un "process ID" o "pid".

Il comando UNIX "ps" elenca tutti i processi in esecuzione in quel momento sulla propria macchina, elencandone anche i pid.

La funzione C:

```
int getpid()
```

restituirà il pid del processo che richiama questa funzione.

Solitamente un programma esegue un singolo processo; comunque più avanti verrà analizzata la maniera di eseguire programmi come parecchi processi separati e comunicanti.

18.04.01. Esecuzione di comandi UNIX da C

È possibile eseguire comandi da un programma C semplicemente come fossero digitati dalla linea di comando UNIX, grazie all'utilizzo della funzione system(). Questo può farci risparmiare molto tempo e molti problemi; infatti in questo modo è possibile eseguire altri programmi di prova, script, e così via, qualora i compiti attualmente svolti ne comportino il lancio.

```
int system(char *string)
```

in cui "string" può essere il nome di una utility unix, uno script shell eseguibile oppure un programma dell'utente. System ritorna lo stato di uscita della shell.

Esempio: chiamata di "ls" da un programma

```
main()
{printf("Files in Directory are: \n");
 system("ls -l");
}
```

"system" è una chiamata composta da 3 altri comandi:

```
execl(), wait() e fork()
```

18.04.01.01. execl()

"execl" ha altre 5 funzioni correlate (vedere le pagine "man"). "execl" stà per "execute and leave", che significa che un processo sarà eseguito e poi terminato dalla stessa execl.

Viene definita con:

```
execl(char *path, char *arg0, ..., char *argn, 0);
```

L'ultimo parametro deve sempre essere 0. È un termine NULL. Poichè la lista degli argomenti è variabile, è necessario avere un modo per dire al C dove termina; il termine nullo serve a questo.

"path" punta al nome di un file contenente un comando che deve essere eseguito, "arg0" punta ad una stringa che è la stessa di "path" (o almeno il suo ultimo componente).

"arg1, ..., argn" sono puntatori agli argomenti per il comando, e 0 segna semplicemente la fine dell'elenco (di lunghezza variabile) degli argomenti.

Così l'esempio in precedenza riportato risulterà ora:

```
main()
{printf("Files in Directory are: \n");
  execl("/bin/ls", "ls", "-l", 0);
}
```

18.04.01.02. fork()

"int fork()" trasforma un singolo processo in due processi identici, riconoscibili come processo padre e processo figlio. In caso di successo, fork() ritorna 0 al processo figlio ed il process ID del processo figlio al processo padre; in caso di esito negativo, fork() ritorna -1 al processo padre, settando errno per indicare l'errore verificatosi, e non viene creato nessun processo figlio.

NOTA: il processo figlio avrà un suo proprio ed unico PID.

Il seguente programma illustra un utilizzo semplice di fork(), dove vengono create due copie del processo ed eseguite assieme (multitasking):

```
main()
{ int return_value;

  printf("Forking process\n");
  fork();
  printf("The process id is %d and return value is %d \n",
        getpid(), return_value);
  execl("/bin/ls", "ls", "-l", 0);
  printf("This line is not printed\n");
}
```

L'output risultante sarà:

```
Forking process
The process id is 6753 and return value is 0
The process id is 6754 and return value is 0
"two lists of files in current directory"
```

NOTA: i processi hanno ID unici, che risulteranno diversi ad ogni esecuzione.

È anche impossibile stabilire in anticipo quale processo utilizzerà il tempo di CPU (così, ogni esecuzione può essere diversa dalla successiva).

Quando vengono generati due processi, possiamo facilmente scoprire (in ogni processo) quale sia il figlio e quale il padre, poichè fork ritorna 0 al figlio.

Possiamo catturare qualsiasi errore se fork ritorna un -1, cioè:

```
int pid; /* process identifier */
```

```

pid=fork();
if (pid < 0)
    {printf("Cannot fork!!\n");
    exit(1);
    }
if (pid == 0)
    {/* child process */ ...}
else
    {/* parent process pid is child's pid */ ...}

```

18.04.01.03. wait()

int wait (int *status_location) - forza un processo padre ad aspettare che un processo figlio si fermi oppure termini.

wait() ritorna il pid del figlio, oppure -1 in caso di errore. Lo stato di uscita del figlio viene ritornato come "status_location".

18.04.01.04. exit()

int exit (int status) - termina il processo che richiama questa funzione e ritorna il valore di uscita "status". Sia i programmi UNIX che quelli C (forked) sono in grado di leggere il valore dello stato. Per convenzione, uno stato uguale a 0 significa "fine normale", mentre qualsiasi altro valore indica un errore o un evento insolito. Molte chiamate alle librerie standard hanno la definizione degli errori nel file header sys/stat.h. Possiamo quindi facilmente derivare le nostre proprie convenzioni.

18.04.02 Utilizzo di pipe in un programma C

Il "piping" è un processo dove l'output di un processo viene trasformato nell'input di un altro. Abbiamo già visto in precedenza esempi di questo dalla linea di comando UNIX, con l'utilizzo di "|". Vedremo ora come è possibile farlo all'interno del programmi C. Avremo due o più processi "forked" che cominceranno fra di loro.

Per prima cosa, è necessario aprire una pipe. UNIX permette di aprire una pipe in due maniere.

18.04.02.01. popen() - Piping formattato

FILE *popen(char *command, char *type) - apre una pipe per I/O dove "command" è il processo che deve essere connesso al processo chiamante, creando così la pipe. Il "type" può essere sia "r" per reading (lettura) che "w" per writing (scrittura).

Il return di popen() è un puntatore ad una stream oppure NULL per un qualsiasi errore.

Una pipe aperta con popen() deve sempre essere chiusa con:

```
pclose(FILE *stream)
```

È possibile comunicare con la "stream" della pipe tramite fprintf() e fscanff().

18.04.02.02. pipe() - Piping di basso livello

int pipe(int fd[2]) - crea una pipe e ritorna due file descrittori, fd[0] e fd[1]. fd[0] è aperto per la lettura, fd[1] per la scrittura.

pipe() ritorna 0 in caso di successo, -1 in caso di fallimento e di conseguenza setta errno.

Il modello standard di programmazione prevede che, dopo la creazione della pipe, due o più processi che cooperano verranno creati da una fork ed i dati verranno passati mediante l'utilizzo di read() e write().

Le pipe aperte con pipe() dovranno essere chiuse con "close (int fd)".

Esempio: il processo padre invia delle write al processo figlio

```
int pdes[2];

pipe(pdes);
if (fork == 0)
    { /* processo figlio */
      close(pdes[1]); /* non richiesto */
      read(pdes[0]); /* legge dal processo padre */
      ...
    }
else
    { close(pdes[0]); /* non richiesto */
      write(pdes[1]); /* scrive al processo figlio */
      ...
    }
```

18.04.03. Interruzioni e segnali

In questa sezione verranno affrontati i modi in cui due processi possono comunicare fra di loro. Quando un processo termina in modo anormale, solitamente prova ad inviare un segnale che indichi cosa è andato a monte. I programmi C (e UNIX) sono in grado di catturare questi segnali ed utilizzarli come diagnostica. Anche le comunicazioni specificate aver luogo in questo modo.

I processi utilizzano dei segnali, che possono essere numerati da 0 a 31. Le macro sono definite nel file header signal.h per quanto riguarda i segnali più comuni.

Queste includono:

```
SIGHUP 1 /* hangup */
SIGINT 2 /* interrupt */
SIGQUIT 3 /* quit */
SIGILL 4 /* illegal instruction */
SIGABRT 6 /* used by abort */
SIGKILL 9 /* hard kill */
SIGALRM 14 /* alarm clock */
SIGCONT 19 /* continue a stopped process */
SIGCHLD 20 /* to parent on child stop or exit */
```

18.04.03.01. Invio di segnali - kill()

int kill(int pid, int signal) - manda un "signal" ad un processo "pid". Se pid è maggiore di 0 il segnale viene inviato al processo il cui process ID corrisponde a pid; se pid è 0, il segnale è mandato a tutti i processi, eccetto i processi di sistema. kill() ritorna 0 per le chiamate che hanno successo, -1 negli altri casi settando conseguentemente "errno". Esiste anche un comando UNIX chiamato kill (vedere le pagine "man").

NOTA: a meno che non si blocchi o venga ignorato, il segnale kill termina il processo. Perciò le protezioni sono incorporate all'interno del sistema.

È possibile eliminare solamente i processi con determinati privilegi di accesso. Una regola di base è quella per cui solamente i processi che appartengono allo stesso utente possono inviare/ricevere messaggi. Il segnale SIGKILL non può bloccarsi o essere ignorato, e terminerà sempre il processo.

Per esempio:

```
kill(getpid(),SIGINT);
```

invierà un segnale di interrupt all'ID del processo chiamante.

Questo avrebbe un effetto simile al comando exit(). Anche CTRL-C, digitato dalla linea di comando, invia un SIGINT al processo correntemente in essere.

unsigned int alarm(unsigned int seconds) - dopo "seconds" secondi, invia il segnale SIGALRM al processo che ha effettuato la chiamata.

18.04.03.02. Ricezione di segnali - signal()

int (*signal(int sig, void (*func)()))() - questo stà ad indicare che la funzione signal() richiamerà la funzione func se il processo riceverà un segnale sig. Signal ritorna un puntatore alla funzione func in caso di successo, oppure ritorna un errore ad errno, e -1 negli altri casi.

func() può avere tre valori:

SIG_DFL - un puntatore alla funzione di default del sistema SIG_DFL(), la quale terminerà il processo al ricevimento di "sig"

SIG_IGN - un puntatore alla funzione di sistema ignore SIG_IGN(), che ignorerà l'azione "sig" (a meno che non sia SIGKILL)

Un indirizzo di funzione - una funzione specificata dall'utente

SIG_DFL e SIG_IGN sono definiti nel file header signal.h (libreria standard).

Così per ignorare un comando CTRL-C dalla linea di comando, dovremo fare:

```
signal(SIGINT, SIG_IGN);
```

Per resettare un sistema, cosicchè SIGINT comporti l'uscita da qualsiasi posizione del nostro programma, dovremo fare:

```
signal(SIGINT, SIG_DFL);
```

Vediamo ora un programma per catturare un CTRL-C, ma non uscire con questo segnale. Abbiamo una funzione "sigproc()" che viene eseguita quando catturiamo in CTRL-C. Possiamo anche settare un'altra funzione per abbandonare il programma se riceve il segnale SIGQUIT, così possiamo terminare il nostro programma:

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

```
void sigproc(void);
```

```

void quitproc(void);

main()
{ signal(SIGINT, sigproc);
  signal(SIGQUIT, quitproc);
  printf("CTRL-C disabled use ctrl-\ to quit \n");
  for(;;) /* loop infinito */ }

void sigproc()
{ signal (SIGINT, sigproc); /* */

/* NOTA: alcune versioni di UNIX resetteranno "signal" al valore di default dopo ogni chiamata. Scosì, per
rispettare la portabilità, facciamo un reset di "signal" ogni volta */

printf("You have pressed CTRL-C \n");
}

void quitproc()
{ printf("ctrl-\ pressed to quit \n");
  exit(0); /* normale status di uscita */
}

```

18.05. Times Up!!

L'ultimo argomento che verrà affrontato in questo corso è quello degli accessi al time del clock con le chiamate di sistema UNIX.

L'utilizzo delle funzioni di time include:

- dire l'ora;
- fornire il tempo a programmi e funzioni;
- settaggio di numeri casuali.

`time_t time(time_t *tloc)` - ritorna il tempo, misurato in secondi, a partire da 00:00:00 GMT, Jan. 1, 1970.

Se "tloc" non è nullo, il valore di ritorno viene anche memorizzato nella locazione a cui punta tloc.

`time()` ritorna il valore del tempo, in caso di successo; in caso di insuccesso, ritorna `(time_t) -1`. "time_t" risulta definito come tipo `long(int)` nei file header `<sys/types.h>` e `<sys/time.h>`.

`int ftimes(struct timeb *tp)` - riempie una struttura puntata da tp, come definito in `<sys/timeb.h>`:

```

struct timeb
{ time_t time;
  unsigned short millitm;
  short timezone;
  short dstflag;
};

```

La struttura contiene il tempo espresso in secondi a partire dall'epoca, con l'intervallo di massima precisione che arriva fino a 1000 millisecondi, il fuso orario locale (misurato in minuti in direzione ovest a partire da Greenwich) ed un flag che, se non è uguale a 0, indica l'ora legale applicata localmente negli appropriati periodi dell'anno. In caso di successo, `ftimes()` non ritorna alcun valore utile, mentre ritorna `-1` in caso di errore.

Altre due funzioni definite in `"#include <time.h>"` sono:

```

char *ctime(time_t *clock)

```

```
char *asctime(struct tm *tm)
```

ctime() converte un long integer (puntato da clock) ad una stringa di 26 caratteri nella forma:

```
Sun Sep 16 01:03:52 1973
```

asctime() ritorna un puntatore alla stringa.

Esempio 1: Tempo (in secondi) per eseguire alcune operazioni:

```
/* timer.c */

#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
{ int i;
  time_t t1,t2;

  (void) time(&t1);
  for (i=1;i<=300;++i)
    printf("%d %d %d \N",1,i*i,i*i*i);
  (void) time(&t2);
  printf("\n Time to do 300 squares and cubes = %d seconds \n",
    (int)t2-t1);
}
```

Esempio 2: Settaggio di un gruppo di numeri casuali:

```
/* random.c */

#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
{ int i;
  time_t t1;

  (void) time(&t1);
  srand48((long)t1);
  /* use time in seconds to set seed */
  printf("5 random numbers
  (Seed = %d):\n", (int)t1);
  for (i=0;i<5;++i)
    printf("%d", lrand48());
  printf("\n\n"); /* flush print buffer */
}
```

lrand48() ritorna dei log integers non negativi uniformemente distribuiti nell'intervallo (0, 2**31).

Una funzione simile drand48() ritorna numeri a doppia precisione compresi nel range [0.0, 1.0].

srand48() setta l'origine per questi generatori di numeri random.

È importante avere gamme diverse quando si richiamano le funzioni, altrimenti verrà generato lo stesso set di numeri pseudo-random.

time() fornisce sempre un'unica gamma d'origine.

19. Opzioni comuni del compilatore C

=====

Qui verranno elencate le opzioni del compilatore C più comuni. Queste possono essere aggiunte alle direttive per il compilatore. Alcune richiedono un argomento aggiuntivo.

Ad esempio:

```
c89 -c -o prog prog.c
```

L'opzione -o richiede un argomento, -c invece no.

19.01. Opzioni di compilazione

- c Sopprime i link con ld(1) e produce un file .o per ogni file sorgente. Ogni singolo file oggetto può essere nominato esplicitamente utilizzando l'opzione -o.
- C Evita che il preprocessore C rimuova i commenti.
- E Esegue il file sorgente solamente attraverso il preprocessore C. Invia l'output allo standard output, oppure ad un file a cui viene dato un nome con l'opzione -o. Include le informazioni di cpp sulla numerazione delle linee (vedere anche l'opzione -P).
- g Produce una tavola informativa dei simboli supplementari per dbx(1) e dbxtool(1). Quando viene data questa opzione, le opzioni -O e -R sono soppresse.
- help Visualizza informazioni di aiuto relative al compilatore.
- Ipathname Aggiunge "pathname" alla lista di directory in cui cercare i file #include con il relativo filename (che non inizi con slash "/"). Il preprocessore cerca i file #include principalmente nella directory contenente il file sorgente, poi nelle directory specificate con l'opzione -I (se ce ne sono) e, alla fine, in /usr/include.
- llibrary Linka con la libreria oggetto "library" (for ld(1)). Questa opzione deve seguire l'argomento del nome del file sorgente.
- Ldirectory Aggiunge "directory" alla lista di directory che contengono routine di librerie oggetto (per il link usando ld(1)).
- M Esegue solamente il preprocessore delle macro sui programmi C elencati, richiedendo che questo generi le dipendenze con il makefile ed invii il risultato sullo standard output (vedere make(1) per dettagli su makefile e sulle regole di dipendenza).
- o outputfile Viene nominato l'output come "outputfile"; quest'ultimo deve avere l'appropriato suffisso per il tipo di file che viene prodotto dalla compilazione (vedere FILES, più avanti). "outputfile" non può essere lo stesso del file sorgente (il compilatore non sovrascriverà il file sorgente).
- O[level] Ottimizza il codice oggetto. Viene ignorato nel caso in cui siano usate le opzioni -g o -a. "-O" omettendo il livello, è equivalente a "-O2". "level" può assumere uno dei seguenti valori:
 - 1 Esegue solamente un'ottimizzazione sul punto critico del livello assembly ;
 - 2 Esegue un'ottimizzazione globale prioritaria sulla generazione del codice, che include l'ottimizzazione dei loop, l'eliminazione delle sottoespressioni comuni, la propagazione delle copie, e l'allocazione automatica nei registri. "-O2" non ottimizza i riferimenti o le definizioni delle variabili esterne o indirette.Se l'ottimizzatore in fase di esecuzione va in "out of memory", questo tenta di eseguire un ripristino riportando la procedura corrente ad un livello più basso di ottimizzazione e riprendendo le procedure successive al livello originale.

- P Esegue il file sorgente solo attraverso il preprocessore C. Mette l'output in un file con un suffisso ".i".
Non include nell'output le informazioni cpp-type sulla numerazione delle linee.

20. Funzioni della libreria standard C

Di seguito troverete elencate quasi tutte le funzioni della libreria standard dell'ANSI C. Nel caso di definizioni relazionate alle funzioni, verrà riportato anche il file header. Questi possono però variare su alcuni sistemi, per cui si consiglia di controllare i manuali di riferimento locali. Viene inclusa una breve descrizione con tutti i tipi dei parametri. Maggiori informazioni possono essere ottenute dalle chiamate online di man o dai manuali di consultazione.

20.01. Manipolazione dei buffer

```
#include <memory.h>
```

void *memchr (void *s, int c, size_t n) - Cerca un carattere in un buffer.

int memcmp (void *s1, void *s2, size_t n) - Paragona due buffers.

void *memcpy (void *dest, void *src, size_t n) - Copia un buffer in un altro.

void *memmove (void *dest, void *src, size_t n) - Sposta un determinato numero di bytes da un buffer ad un altro.

void *memset (void *s, int c, size_t n) - Setta tutti i bytes di un buffer ad un dato carattere.

20.02. Classificazione dei caratteri e conversione

```
#include <ctype.h>
```

int isalnum(int c) -Vero se "c" è alfanumerico.

int isalpha(int c) - Vero se "c" è una lettera dell'alfabeto.

int isascii(int c) - Vero se "c" è ASCII .

int iscntrl(int c) - Vero se "c" è un carattere di controllo.

int isdigit(int c) -Vero se "c" è un numero decimale.

int isgraph(int c) - Vero se "c" è un carattere grafico.

int islower(int c) - Vero se "c" è una lettera minuscola.

int isprint(int c) - Vero se "c" è un carattere stampabile.

int ispunct (int c) - Vero se "c" è un carattere di punteggiatura.

int isspace(int c) - Vero se "c" è un carattere spazio.

int isupper(int c) - Vero se "c" è una lettera maiuscola.

int isxdigit(int c) - Vero se "c" è un numero esadecimale.

int toascii(int c) - Converte "c" in ASCII.

tolower(int c) - Converta "c" in minuscolo.

int toupper(int c) - Converta "c" in maiuscolo.

20.03. Conversione dei dati

#include <stdlib.h>

double atof(char *string) - Converta una stringa in un valore in floating point.

int atoi(char *string) - Converta una stringa in un valore integer.

int atol(char *string) - Converta una stringa in un valore long integer.

char *itoa(int value, char *string, int radix) - Converta un valore integer in una stringa utilizzando il "radix" dato.

char *ltoa(long value, char *string, int radix) - Converta un valore long integer in una stringa in un dato "radix".

double strtod(char *string, char *endptr) - Converta una stringa in un valore in floating point.

long strtol(char *string, char *endptr, int radix) - Converta una stringa in un valore long integer utilizzando un dato "radix".

unsigned long strtoul(char *string, char *endptr, int radix) - Converta una stringa in un valore long senza segno.

20.04. Manipolazione delle directory

#include <dir.h>

int chdir(char *path) - Cambia la directory corrente a quella contenuta nel "path" dato.

char *getcwd(char *path, int numchars) - Ritorna il nome della directory di lavoro corrente.

int mkdir(char *path) - Crea una directory utilizzando il nome "path" specificato.

int rmdir(char *path) - Rimuove la directory specificata.

20.05. Manipolazione dei file

#include <sys/stat.h> e #include <sys/types.h>

int chmod(char *path, int pmode) - Cambia il settaggio dei permessi di un file.

int fstat(int handle, struct stat *buffer) - Ottiene informazioni sul file status.

int remove(char *path) - Cancella il file indicato.

int rename(char *oldname, char *newname) - Rinomina un file.

int stat(char *path, struct stat *buffer) - Ottiene informazioni sul file status del file indicato.
unsigned umask(unsigned pmode) - Setta il mask dei permessi di un file.

20.06. Input e Output

20.06.01. Stream I/O

#include <stdio.h>

void clearerr(FILE *file_pointer) - Cancella l'indicatore di errore della stream.

int fclose(FILE *file_pointer) - Chiude un file.

int feof(FILE *file_pointer) - Controlla se è stato incontrato un end-of-file in una stream.

int ferror(FILE *file_pointer) - Controlla se è stato riscontrato un qualsiasi errore durante il file di I/O.

int fflush(FILE *file_pointer) - Scarica (flush) il buffer in un file.

int fgetc(FILE *file_pointer) - Prende un carattere da una stream.

int fgetpos(FILE *file_pointer, fpos_t current_pos) - Ottiene la posizione corrente all'interno di una stream.

char *fgets(char *string, int maxchar, FILE *file_pointer) - Legge una stringa da un file.

FILE *fopen(char *filename, char *access_mode) - Apre un file per l'I/O bufferizzato.

int fprintf(FILE *file_pointer, char *format_string, args) - Scrive output formattato in un file.

int fputc(int c, FILE *file_pointer) - Scrive un carattere in una stream.

int fputchar(int c) - Scrive un carattere sullo "stdout".

int fputs(char *string, FILE *file_pointer) - Scrive una stringa in una stream.

size_t fread(char *buffer, size_t size size_t count, FILE *file_pointer) - Legge dati non formattati da una stream in un buffer.

FILE *freopen(char *filename, char *access mode, FILE *file_pointer) - Riassegna un file puntatore ad un file diverso.

int fscanf(FILE *file_pointer, char *format string, args) - Legge input formattato da una "stream".

int fseek(FILE *file_pointer, long offset, int origin) - Setta la posizione corrente nel file ad una nuova locazione.

int fsetpos(FILE *file pointer, fpos_t *current pos) - Setta la posizione corrente nel file ad una nuova locazione.

long ftell(FILE *file_pointer) - Ottiene la locazione corrente nel file.

size_t fwrite(char *buffer, size_t size, size_t count FILE *file_pointer) - Scrive dati non formattati da un buffer ad una stream.

int getc(FILE *file_pointer) - Legge un carattere da una "stream".

int getchar(void) - Legge un carattere da "stdin".

char *gets(char *buffer) - Legge una linea da "stdin" in un buffer.

int printf(char *format_string, args) - Scrive output formattato su "stdout".

int putc(int c, FILE *file_pointer) - Scrive un carattere in una "stream".

int putchar(int c) - Scrive un carattere su "stdout".

int puts(char *string) - Scrive una stringa su "stdout".

void rewind(FILE *file_pointer) - Esegue il rewind di un file.

int scanf(char *format_string, args) - Legge input formattato da "stdin".

void setbuf(FILE *file_pointer, char *buffer) - Costruisce un nuovo buffer per la stream.

int setvbuf(FILE *file_pointer, char *buffer, int buf_type, size_t buf size) - Costruisce un nuovo buffer e controlla il livello di bufferizzazione in una stream.

int sprintf(char *string, char *format_string, args) - Scrive output formattato su una "string".

int sscanf(char *buffer, char *format_string, args) - Legge input formattato da una "string".

FILE *tmpfile(void) - Apre un file temporaneo.

char *tmpnam(char *file_name) - Ottiene un file name temporaneo.

int ungetc(int c, FILE *file_pointer) - Respinge un carattere nel buffer della "stream".

20.06.02. I/O di basso livello

#include <stdio.h> e possono essere necessari anche alcuni dei seguenti:

#include <stdarg.h>, #include <sys/types.h>,
#include <sys/stat.h>, #include <fcntl.h>.

int close (int handle) - Chiude un file aperto per I/O non bufferizzato.

int creat(char *filename, int pmode) - Crea un nuovo file con il settaggio dei permessi come specificato.

int eof (int handle) - Controlla l'end-of-file.

long lseek(int handle, long offset, int origin) - Va ad una specifica posizione in un file.
int open(char *filename, int oflag, unsigned pmode) - Apre un file per I/O di basso livello.
int read(int handle, char *buffer, unsigned length) - Legge dati binari da un file ad un buffer.
int Write(int handle, char *buffer, unsigned count) - Scrive dati binari da un buffer ad un file.

20.07. Matematica

#include <math.h>

int abs (int n) - Get absolute value of an integer.
double acos(double x) - Compute arc cosine of x.
double asin(double x) - Compute arc sine of x.
double atan(double x) - Compute arc tangent of x.
double atan2(double y, double x) - Compute arc tangent of y/x.
double ceil(double x) - Get smallest integral value that exceeds x.
double cos(double x) - Compute cosine of angle in radians.
double cosh(double x) - Compute the hyperbolic cosine of x.
div_t div(int number, int denom) - Divide one integer by another.
double exp(double x) - Compute exponential of x.
double fabs (double x) - Compute absolute value of x.
double floor(double x) - Get largest integral value less than x.
double fmod(double x, double y) - Divide x by y with integral quotient and return remainder.
double frexp(double x, int *expPtr) - Breaks down x into mantissa and exponent of no.
labs(long n) - Find absolute value of long integer n.
double ldexp(double x, int exp) - Reconstructs x out of mantissa and exponent of two.
ldiv_t ldiv(long number, long denom) - Divide one long integer by another.
double log(double x) - Compute log(x).
double log10 (double x) - Compute log to the base 10 of x.
double modf(double x, double *intPtr) - Breaks x into fractional and integer parts.
double pow (double x, double y) - Compute x raised to the power y.
int rand (void) - Get a random integer between 0 and 32.
int random(int max_num) - Get a random integer between 0 and max_num.

<code>void randomize(void)</code>	- Set a random seed for the random number generator.
<code>double sin(double x)</code>	- Compute sine of angle in radians.
<code>double sinh(double x)</code>	- Compute the hyperbolic sine of x.
<code>double sqrt(double x)</code>	- Compute the square root of x.
<code>void srand(unsigned seed)</code>	- Set a new seed for the random number generator (rand).
<code>double tan(double x)</code>	- Compute tangent of angle in radians.
<code>double tanh(double x)</code>	- Compute the hyperbolic tangent of x.

20.08. Allocazione di memoria

`#include <malloc.h>`

<code>void *calloc(size_t num elems, size_t elem_size)</code>	- Alloca un vettore ed inizializza tutti gli elementi a zero.
<code>void free(void *mem address)</code>	- Libera un blocco di memoria.
<code>void *malloc(size_t num bytes)</code>	- Alloca un blocco di memoria.
<code>void *realloc(void *mem address, size_t new size)</code>	- Alloca nuovamente (aggiustando la dimensione) un blocco di memoria.

20.09. Controllo dei processi

`#include <stdlib.h>`

<code>void abort(void)</code>	- Interrompe l'esecuzione di un processo.
<code>int execl(char *path, char *arg0, char *arg1, ..., NULL)</code>	- Lancia un processo figlio (passaggio della linea di comando).
<code>int execlp(char *path, char *arg0, char *arg1, ..., NULL)</code>	- Lancia un processo figlio (utilizzando PATH, passaggio della linea di comando).
<code>int execv(char *path, char *argv[])</code>	- Lancio di un processo figlio (passaggio del vettore argument).

<code>int execlp(char *path, char *argv[])</code>	- Lancio di un processo figlio (utilizzando PATH, passaggio del vettore argument).
<code>void exit(int status)</code>	- Termina un processo dopo aver svuotato tutti i buffers.
<code>char *getenv(char *varname)</code>	- Ottiene la definizione di una variabile di environment.
<code>void perror(char *string)</code>	- Stampa il messaggio di errore corrispondente all'ultimo errore di sistema.
<code>int putenv(char *envstring)</code>	- Inserisce una nuova definizione nella environment table.
<code>int raise(int signum)</code>	- Genera un segnale C (exception).
<code>void (*signal(int signum, void(*func)(int signum [, int subcode])))(int signum)</code>	- Stabilisce un signal handler per il numero signal "signum".
<code>int system(char *string)</code>	- Esegue un comando UNIX (o comunque del sistema operativo residente).

20.10. Ricerca e ordinamento

`#include <stdlib.h>`

`void *bsearch(void *key, void *base, size_t num, size_t width, int (*compare)(void *elem1, void *elem2))`

- Esegue una ricerca binaria.

`void qsort(void *base, size_t num, size_t width, int (*compare)(void *elem1, void *elem2))`

- Utilizza l'algoritmo di ordinamento veloce per ordinare un vettore.

20.11. Manipolazione di stringhe

`#include <string.h>`

`char *strcpy(char *dest, char *src)`

- Copia una stringa in un'altra.

`int strcmp(char *string1, char *string2)`

- Confronta string1 e string2 per determinare l'ordine alfabetico.

`char *strncpy(char *string1, char *string2)`

- Copia string2 in string1.

`char *strerror(int errnum)`

- Ottiene il messaggio di errore corrispondente al numero di errore specificato.

`int strlen(char *string)`

- Determina la lunghezza di una stringa.

char *strncat(char *string1, char *string2, size_t n) - Aggiunge "n" caratteri di string2 in string1.
int strncmp(char *string1, char *string2, size_t n) - Confronta i primi "n" caratteri di due stringhe.
char *strncpy(char *string1, char *string2, size_t n) - Copia i primi "n" caratteri di string2 in string1.
char *strnset(char *string, int c, size_t n) - Setta i primi "n" caratteri di string a "c".
char *strchr(char *string, int c) - Cerca l'ultima occorrenza del carattere "c" in string.

20.12. Time

#include <time.h>

char *asctime (struct tm *time) - Converte time da "struct tm" a stringa.
clock_t clock(void) - Ottiene l'elapsed processor time in clock ticks.
char *ctime(time_t *time) - Converte il binary time in una stringa.
double difftime(time_t time2, time_t time1) - Calcola la differenza in secondi tra due orari.
struct_tm *gmtime (time_t *time) - Ottiene il Greenwich Mean Time (GMT) in una "tm structure".
struct tm *localtime(time_t *time) - Ottiene l'orario locale in una "tm structure".
time_t time(time_t *timeptr) - Ottiene l'orario corrente come secondi trascorsi dall'ora 0 GMT 1/1/70.