Scritto delle lezioni del Corso di Laboratorio di Programmazione

prof. Josef Eschgfäller

Mantovani Filippo
Dipartimento di Matematica - Università di Ferrara
http:\\members.xoom.it\filimanto

Indice

CAPITOLO 1

Il programma minimo

1. Il programma minimo in C

```
// alfa.c
 # include <stdio.h>
 int main();

////////////

 int main()
 {puts ("ciao");
 exit(0);}
```

Creiamo una cartella `c` e inseriamo questo file `alfa.c`. Per trasformarlo in un file oggetto usiamo il comando `gcc -c alfa.c` si crea così il file `alfa.o` controlliamo con `ls` l'esistenza del file oggetto `alfa.o`. Questo file non è ancora eseguibile. Creiamo un file eseguibile con il comando
`gcc -o alfa alfa.o -lc -lm`. (∗)
L'indicazione `-lc -lm` significa che nella composizione del programma eseguibile `alfa` vengono utilizzate la libreria base del C (ad essa si riferisce `-lc` e la libreria matematica `-lm`). Queste librerie sono fisicamente contenute nei files `libc.so` e `libm.so` della cartella `/usr/lib`. Le librerie `.so` sono dinamiche, non vengono cioè aggiunte al programma eseguibile `alfa`, ma caricate in memoria separatamente. Esistono anche le librerie statiche `libc.a` e `libm.a` nella stessa cartella che possono essere usate per creare un programma indipendente dalle librerie dinamiche.

In verità `libm.so` è un link simbolico a una versione precisa della libreria nel nostro caso a `libmp.so 3.1.4`. Le librerie si possono trovare anche in altre cartelle, ad esempio la *gnu scientific library (GSL)* e la sottolibreria riferita alla versione C del *BLAS (Basic Linear Algebra Subprograms)* vengono tipicamente installate nella cartella `usr/local/bin` con i nomi `libgcl.so` e `libgslcblas.so`, rispettivamente `libgsl.a` e `libgslcblas.a`. Se le vogliamo usare nel comando (*) dobbiamo aggiungere `-lgsl -lgslcblas`, in altre parole il prefisso `lib` diventa `l` e il suffisso `.so` e `.a` viene omesso.

2. Librerie statiche e dinamiche

Il file sorgente `alfa.c` occupa solo 145 byte e il file oggetto è grande 832 byte. Il programma eseguibile `alfa` che abbiamo creato con il comando (*) del paragrafo precedente occupa 11507 byte.

Se avessimo invece utilizzato le librerie statiche con l'opzione `-static` con il comando di composizione (=*link*) `gcc -o alfa alfa.o -lc -lm -static` lasciando invariante il comando di compilazione avremmo ottenuto un programma `alfa` di 447048 byte cioè molto più grande. Esso contiene infatti adesso il codice delle librerie del C e delle funzioni matematiche al suo interno.

Useremo sempre le librerie dinamiche omettendo cioè `-static`. Si hanno così i seguenti

vantaggi: il programma rimane molto più piccolo; le librerie dinamiche vengono solo caricate in memoria centrale se non sono già state caricate da un altro programma; correzioni in una delle librerie non richiedono in genere la ricompilazione dei programmi che le usano, tranne nel caso in cui si utilizzino nuove funzioni.

svantaggi: trasportando un programma sul nuovo computer bisogna trasportare separatamente anche le librerie dinamiche se non ci sono già; eventualmente incompatibilità delle versioni.

Nel comando di composizione `-o alfa` significa che viene creato un file eseguibile di nome `alfa`. Se avessimo anche altri file sorgente `beta.c` e `gamma.c` da compilare dovremmo battere i seguenti comandi per compilarli:

```
gcc -c alfa.c
gcc -c beta.c
gcc -c gamma.c
```

e il comando

```
gcc -o alfa alfa.o beta.o gamma.o -lc -lm
```

per la composizione e la creazione del file eseguibile `alfa`.

Per evitare questo lavoro sotto Linux si può scrivere un *makefile* chiamato Makefile.

```
# Makefile

lib = -lc -lm
obj = Oggetti
VPATH = ${obj}

make: alfa.o
    gcc -o alfa ${obj}/*.o ${lib}

%.o:%.c
    gcc -o ${obj}/$*.o -fwritable -strings $*.c

PHONY: clean
clean:
    rm -f *.o ${obj}/*.o
```

Affinchè questo makefile funzioni bisogna creare una cartella `Oggetti` dove verranno raccolti i files `.o`.

Se vogliamo lavorare con altre sorgenti `beta.c` e `gamma.c` nella riga che comincia con `make:` scriviamo `beta.o gamma.o` e poi `make` dalla shell.

Una volta composto, il programma può essere utilizzato digitando `alfa` dalla shell.

3. Dettaglio del programma del paragrafo 1

1. `//` è un commento e in questo caso riporta il nome del file (utile per la stampa).
2. `# include <gino>` fa in modo che il preprocessore (che prepara il file per il compilatore) inserisca in questo punto il file `alfa` che si deve trovare in una delle cartelle dove il compilatore cerca questi file da includere (soprattutto `/usr/include`). Con `# include "gino"` viene invece incluso il file (assoluto o locale secondo la convenzione Unix) il cui nome è proprio `gino`. Nel nostro caso con `# include <stdio.h>` viene incluso il file che contiene le dichiarazioni delle funzioni delle variabili standard di input/output del C (ad esempio `printf` e `puts`).
3. `int main();` è la **dichiarazione** della funzione principale del programma. Ogni programma in C deve contenere, anche se è disposto su più file, esattamente una funzione principale che deve portare il nome di `main` e il cui tipo di risultato (che precede il nome della funzione) è `int` (l'intero più grande sui calcolatori attuali è circa pari a 2^{32}).
4. `int main() {...}` è la la **definizione** della funzione `main`. Il corpo della funzione è racchiuso tra parentesi graffe.
5. `puts("alfa");` stampa la stringa alfa sullo schermo (`puts` è un'abbreviazione di *put string*) con un carattere di invio alla fine. Si potrebbe anche usare `printf("alfa\n");`. `printf` permette un output formattato molto più generale (come vedremo in seguito).
6. Sotto Unix i processi attivi comunicano continuamente tra di loro; `exit(0);` comunica agli altri processi che il programma è terminato senza errori.

I file `.c` e `.h` sono normali file di testo (in formato testo puro). Per scriverli si può usare un editor che crea files in formato testo puro (ad esempio Emacs o Kate - KDE Advanced Text Editor - presente in molte distribuzioni Linux oppure su kate.kde.org).

Invece di `# include <stdio.h>` nel sorgente `alfa.c` stesso, è preferibile, soprattutto quando si vogliono usare più file `.c`, creare un file `alfa.h` nella nostra cartella e trasferire in esso le istruzione di inclusione generali. In `alfa.c` scriviamo allora `# include "alfa.h"`. Abbiamo quindi nel file `alfa.h`:

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

e nel file `alfa.c`:

```
 //alfa.c
 # include "alfa.h"

 int main();
 ///////////////

 int main()
 {puts ("ciao");
```

```
exit(0);}
```

Il C standard prevede molti altri file di intestazione di cui inseriamo adesso i seguenti in `alfa.h`:

```
// alfa.h
# include <ctype.h>
# include <complex.h>
# include <dirent.h>
# include <fcntl.h>
# include <limits.h>
# include <locale.h>
# include <math.h>
# include <signal.h>
# include <stdarg.h>
# include <stdio.h>
# include <string.h>
# include <sys/stat.h>
# include <sys/times.h>
# include <sys/types.h>
# include <time.h>
# include <unistd.h>
```

Le più importanti sono:
`stdio.h` che riguarda l'input/output,
`stdlib.h` per le funzioni generali di utilità,
`math.h` per le funzioni matematiche,
`string.h` per stringhe e altre funzioni matematiche,
`sys/stat.h` e `dirent.h` per le proprietà di files e cartelle,
`stdarg.h` per le funzioni con un numero variabile di argomenti,
`complex.h` che esiste solo da pochi anni e aggangia le funzioni per i numeri complessi (che una volta non esistevano in C).

In genere il file che contiene la `main` (noi lo chiamiamo sempre `alfa.c`) oltre alla `main` contiene poche altre funzioni. La maggior parte delle funzioni del programma vengono raccolte negli altri file.

CAPITOLO 2

Istruzioni di controllo

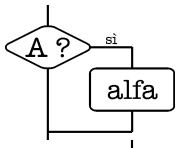
Il C conosce tre tipi di istruzioni di controllo:

i. Istruzioni condizionali: `if ... else`, `switch`.
ii. Cicli: `for`, `while`, `do ... while`.
iii. Salti: `goto`

1. if ... else

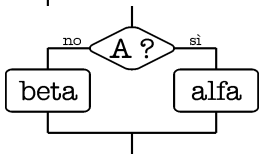
1. `if(A) alfa;`

   corrisponde al diagramma di flusso:



2. `if(A) alfa; else beta;`

   corrisponde al diagramma di flusso:



Calcoliamo ad esempio il massimo tra due numeri reali (a cui nel C corrisponde il tipo `double`) con la seguente funzione:

```
double Max (double a, double b)
{if (a<b) return b; return a;}
```

È una buona abitudine usare per le proprie funzioni nomi che iniziano con una lettera maiuscola; ciò non solo evita molte interferenze con i nomi di funzioni di sistema, ma rende anche l'organizzazione dei programmi esteticamente più trasparente.

Si osservi che in questo caso non abbiamo usato `else`; infatti se a<b si esce in ogni caso dalla funzione.

Attenzione: forse l'unico veramente fastidioso difetto del C è che, quando un argomento di una funzione è dichiarato come `double` interi che vengono usati per questo argomento devono essere riconoscibili come `double`. Quindi `x=Max(5,8);` non è corretto, ma bisogna usare `x=Max(5.0,8.0);` oppure, soprattutto quando gli elementi sono variabili, una conversione di tipo:

```
int a=3; int b=7;
x=Max((double)a,(double)b);
```

In modo simile definiamo una funzione per il segno di un numero reale:

$$x = \begin{cases} 1 & \text{se } x > 0 \\ 0 & \text{se } x = 0 \\ -1 & \text{se } x < 0 \end{cases}$$

```
int Leqno(double x)
{if (x>0) return 1; if (x==0) return 0; return -1;}
```

Il problema del `3x+1`**.** Questo è sicuramente il problema più inutile della matematica; sembra incredibile che, da quando è stato posto più di 60 anni fa, nessuno sia riuscito a risolverlo.

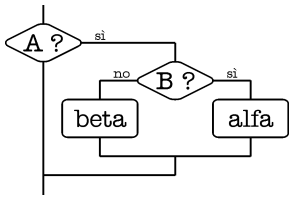
Partiamo con un numero naturale qualsiasi maggiore di 1; se è pari lo dividiamo per 2 altrimenti lo moltiplichiamo per 3 e aggiungiamo 1. Questa operazione T è descritta dalla seguente funzione:

```
int T (int x)
{if (x%2==0) return x/2; return 3*x+1;}
```

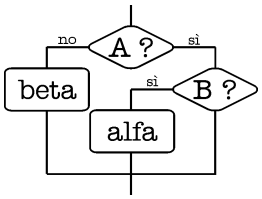
Con il numero così ottenuto ripetiamo l'operazione e ci fermiamo solo quando arriviamo a 1. Ad esempio, partendo con 7 otteniamo 22, poi 11, poi 34, poi 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Sembra che alla fine l'algoritmo si fermi sempre, cioè che prima o poi si arrivi sempre ad 1, ma nessuno riesce a dimostrarlo.

L'`if` può essere annidato; ogni `else` è associato all'`if` che lo precede più da vicino e che non possiede già un `else`:

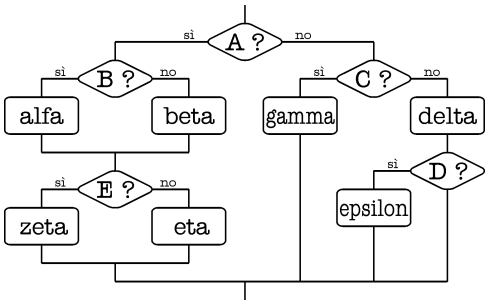
1. `if(A) if(B) alfa; else beta;`



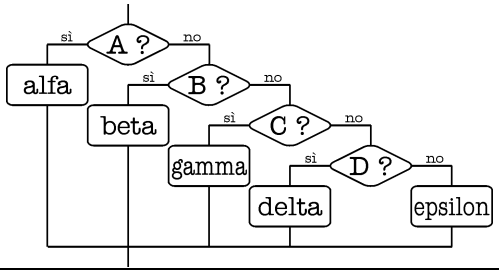
2. `if(A) {if(B) alfa;} else beta;`



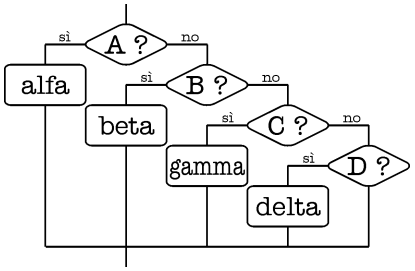
3. `if(A)`
   `{if(B) alfa; else beta;`
   `if(E) zeta; else eta;}`
   `if(C) gamma;`
   `else {delta; if(D) epsilon;}`



4. `if(A) alfa; else`
   `if(B) beta; else`
   `if(C) gamma; else`
   `if(D) delta; else`
   `epsilon;`



5.
```
if(A) alfa; else
  if(B) beta; else
  if(C) gamma; else
  if(D) delta;
```

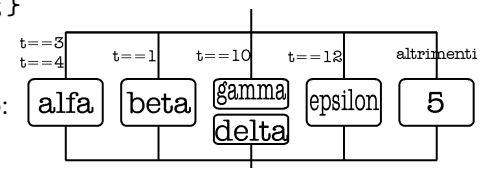


2. switch

L'istruzione

```
switch(t) {case 3: case 4: alfa;
case 1: beta; break;
case 10: gamma; delta; break;
case 12: epsilon; default: eta;}
```

corrisponde al diagramma di flusso:

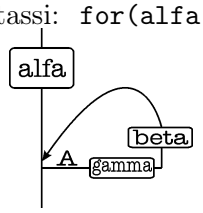


`t` deve essere un'*espressione* di tipo intero o compatibile con il tipo intero; i valori che seguono i `case` devono essere *costanti* di tipo compatibile con il tipo intero; queste costanti devono essere tutte distinte.

Attenzione: i prefissi `case` e `default` non alterano il flusso di controllo che continua liberamente attraverso i prefissi finchè non si incontra un `break`. I `case` e i `default` nello `switch` del C hanno infatti il significato di etichette e senza il `break` non si escludono a vicenda. Il `default` può anche mancare e infine gli `switch` possono essere annidati.

3. for

Il `for` del C ha la sintassi: `for(alfa;A;beta) gamma;` equivalente al diagramma di flusso:

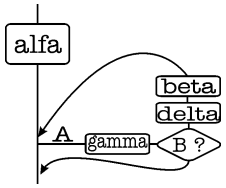


`alfa` e `beta` sono successioni di istruzioni separate da virgole; l'ordine in cui le istruzioni in ogni successione vengono eseguite non è prevedibile; ciascuno dei tre campi può essere anche vuoto.

Per uscire da un ciclo `for` si può usare il comando `break`:

```
for(alfa;A;B) {gamma; if(B) break; delta;}
```

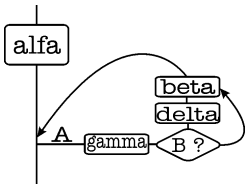
Questo corrisponde al diagramma di flusso:



L'istruzione `continue` interrompe il passaggio corrente di un ciclo e porta all'immediata esecuzione del passaggio successivo. Un esempio:

```
for(alfa;A;B)
{gamma; if(B) continue; delta;}
```

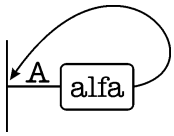
e corrisponde al diagramma di flusso:



`break` e `continue` possono essere utilizzate con lo stesso significato anche nel `while` e nel `do ... while`.

4. while

L'istruzione `while(A) alfa;` è equivalente a `for(;A;) alfa;` e corrisponde al diagramma di flusso:



La seguente funzione stampa sullo schermo, x, Tx, T^2x, fino a $Tx = 1$ con T definita come in 1

```
void Passi3xp1 (int x)
{while(x>1)
 {printf("%d ",x); x=T(x);}}
```

Il programma completo.

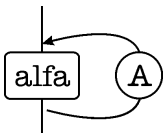
```
//alfa.c
# include "alfa.h"

void Passi3xp1(), Prova();
int T();

int main();
/////////
int main()
{Prova();
exit(0);}
/////////

void Passi3xp1 (int x)
{while(x>1) {printf("%d ",x); x=T(x);}}

void Prova()
{int x;
for(x=7;x<=12;x++)
{Passi3xp1(x); printf("\n");}}
int T (int x)
{if(x%2==0)return x/2; return 3*x+1;}

// Output: 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2
```

```
8 4 2
9 29 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2
10 5 16 8 4 2
11 34 17 52 26 13 40 20 10 5 16 8 4 2
12 6 3 10 5 16 8 4 2
```

5. do ... while

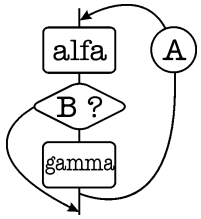
La sintassi è `do alfa; while (A);`

e il diagramma di flusso relativo è:



`do {alfa; if(B) break; gamma;} while(A);`

ha invece questo diagramma di flusso:



E' facile confondere il `while` finale di un `do ... while` con un `while` iniziale con istruzione vuota del tipo `while(A);` infatti la differenza consiste solo nel fatto che nel primo caso il `while` è preceduto da un `do`.

Il `for` del C è molto generale e comprende sia il `while` che (con un piccolo artificio) il `do ... while`.
`while(A) alfa;` è equivalente a: `for(;A;) alfa;`
`do alfa while (A)` è equivalente a: `alfa; while(A) alfa;` e quindi a `alfa; for(;A;) alfa;`

`alfa` però può essere un'istruzione complessa e per non doverla ripetere si può usare questo trucco:

```
int u;
for(u=1;A||u;u=0) alfa;
```

con il piccolo difetto che l'istruzione ausiliaria `u=0` viene eseguita in ogni passo del ciclo.

In `if(A)` o `for(;A;)` non abbiamo ancora specificato in cosa deve consistere l'espressione `A` che descrive la condizione che deve essere soddisfatta. Il C non prevede variabili o espressioni booleane esplicite e usa 0 per falso, mentre ogni numero diverso da 0 è considerato vero. Nei confronti si usano minore (`<`), minore uguale (`<=`), maggiore (`>`), maggiore uguale (`>=`), per il confronto si deve usare il doppio segno di uguale (`==`) e per la non uguaglianza il simbolo `!=`.

Attenzione: il C accetta anche l'istruzione `if(a=1)`, ma in questo caso prima viene assegnato ad `a` il valore uguale ad 1 e questo valore poi viene usato per il test, quindi in questo caso la condizione `if` è sempre vera (e non ha senso). Mentre anche il numero 2.73 è considerato vero, il risultato di

un'espressione booleana che usa operatori di confronto o operatori logici è sempre uguale a 0 o 1, quindi la funzione

```
int maggioredelquad(int a, int b)
{return a>b*b;}
```

dà come valore 1 se $a > b^2$ e 0 altrimenti.

Gli operatori logici sono `&&` per indicare `AND`, `||` per `OR` e `!` per `NOT`.

Nel C99 esiste anche la possibilità di usare il tipo `bool` con in valori `true` e `false` includendo `<stdbool.h>` e altri nomi per gli operatori logici come `and` per `&&`, `bitand` per `&` ecc. includendo `<iso646.h>`.

6. goto

L'uso del `goto` richiede un'etichetta che è un identificatore seguito da doppio punto. L'etichetta deve trovarsi nella stessa funzione in cui è usata; può essere locale in un blocco. Il `goto` nella programmazione moderna viene in genere evitato essendo un tipico costrutto della programmazione basata sui diagrammi di flusso. E' però utile nel caso di più cicli annidati, infatti il `break` esce solo dal ciclo in cui si trova.

Esempio:

```
while(A)
 {alfa; for(...)
   {beta; while(B)
     {gamma; if(C) goto fine; delta;}
     epsilon;}
   zeta;}
 fine: eta;}
```

7. ? e ,

Questa costruzione del C (presente anche nel Perl) può essere talvolta usata per la distinzione di casi al posto di un `if ... else` o di uno `switch`.

L'espressione `A ? u : v` è un valore che è uguale ad `u` se `A` è soddisfatta altrimenti uguale a `v`. L'istruzione `x=A ? u : v` è quindi equivalente a `if(A) x=u, else x=v;`. Queste costruzioni possono essere annidate come nel seguente esempio:

```
int Segno (double x)
{return x>0 ? 1 : x<0 ? -1 : 0;}
```

Spesso il punto interrogativo viene combinato con l'operatore virgola (`,`): `(alfa, beta, gamma, u)` è un valore che è uguale a `u` dopo l'esecuzione nell'ordine indicato di `alfa`, `beta`, `gamma`, `delta`. `x=(alfa, beta, gamma, u);` è equivalente quindi a `alfa; beta; gamma; x=u;`. Evidentemente `?` e `,` possono essere sostituiti con gli operatori visti in precedenza. Risultano però utili in situazioni con molte distinzioni di casi.

Esempio: Presentiamo un algoritmo binario per il massimo comune divisore che talvolta viene utilizzato al posto dell'algoritmo euclideo. Siano $a, b \in Z$;
se a, b sono entrambi pari allora $\mathrm{mcd}(a, b) = 2\mathrm{mcd}(a/2, b/2)$;
se a è pari e b è dispari allora $\mathrm{mcd}(a, b) = \mathrm{mcd}(a/2, b)$;

se a è dispari e b è pari allora mcd(a,b) =mcd$(a,b/2)$;
se a e b sono entrambi dispari allora mcd(a,b) =mcd$((a-b)/2,b)$;
$a \geq 0 \Rightarrow \text{mcd}(a,0) = a$
$b \geq 0 \Rightarrow \text{mcd}(b,0) = b$
mcd$(-a,b)$ =mcd(a,b)
mcd$(a,-b)$ =mcd(a,b)

```
int Mcd(int a, int b)
{int apari, bpari;
return a<0 ? mcd(-a,b) : b<0 ? mcd(a,-b) : a==0 ? b : b==0 ? a :
 (apari=(a%2==0), bpari=(b%2==0,
 apari && bpari ? 2*mcd(a/2,b/2) :
 apari ? mcd(a/2,b) : bpari ? mcd(a,b/2) :
 a<b ? mcd(b,a) : mcd(b,a) : mcd((a-b)/2,b));}
```

Nell'ultima riga non bisogna dimenticare di invertire l'ordine degli argomenti altrimenti l'algoritmo non termina. Senza l'inversione infatti la coppia (3,17) verrebbe elaborata così: (3,17) → (-7,17) → (7,17) → (5,17) → (-6,17) → (6,17) → (3,17). (Per esercizio si può provare a riscrivere la funzione usando `if ... else`).

Più semplice da programmare è l'algoritmo euclideo che usa la relazione:

$$mcd(a,b) = \begin{cases} a & \text{se} \quad b = 0 \text{ e } a \geq 0 \\ mcd(b, a\%b) & \text{se} \quad b > 0 \end{cases}$$

```
int Mcd(int a, int b)
 {return a<0 ? Mcd(-a,b) : b<0 ? Mcd(a,-b) : b ? Mod(b,a%b) : a;}
```

Probabilmente si preferirebbe eseguire le operazioni per rendere `a` e `b` positivi indipendentemente dalla ricorsione, quindi:

```
int Mcd(int a, int b)
{if(a<0) a=-a; if(b<0) b=-b;
return b ? Mcd(b,a%b) : a;}
```

Se sappiamo già in anticipo che `a` e `b` non sono mai negativi, basta addirittura:

```
int Mcd(int a, int b)
return b ? Mcd(b,a%b) : a;}
```

Per controllare possiamo usare:

```
int main()
{int n;
for (n=2; n<20; n++) printf("%d %d\n,n,Mcd(n,12));
printf("%d\n",Mcd(7464,3580));
exit(0);}
```

CAPITOLO 3

I tipi di dati elementari

1. Tipi standard

Esistono formalmente numerose variazioni di tipi di dati elementari; in pratica però sono quasi sempre sufficienti i tipi `char`, `int`, `unsigned int`, `long`, `site_t` (e simili tipi speciali come `pid_t` o `uid_t`), `double`, `void`.

Vettori e puntatori e tipi composti (strutture ed unioni) verranno trattati più avanti.

Il tipo `char` viene utilizzato per rappresentare caratteri e corrisponde quasi sempre ad un numero compreso tra 0 e 255 a cui può essere convertito. I caratteri *americani* corrispondono da 0 a 227 e sono codificati nella tabella ASCII.

0-31	caratteri speciali (tasti di controllo)
32	(spazio)
33-47	`! " # $ % & ' ( ) * + , - . /`
48-57	0 1 2 3 4 5 6 7 8 9
58-64	`: ; < = > ? @`
65-90	A ... Z
91-96	`[ \ ] ^ _ '`
97-122	a ... z
123-126	`{ | } ~`
127	Cancella un carattere

Il tipo `int` sui pc rappresenta interi tipicamente compresi tra -2.147.483.648 e 2.147.483.647 (-2^{31} e $2^{31}-1$); `unsigned int` interi maggiori o uguali a zero, quindi compresi tra 0 e $2^{32}-1 = 4.294.967.296$. Attualmente `long` è spesso uguale a `int` e `size_t` uguale a `long`.

Il tipo `double` rappresenta numeri reali naturalmente anch'essi entro limiti di grandezza e di precisione.

Il numero di byte occupato da un tipo o da una variabile di quel tipo può essere ottenuto con la funzione `sizeof`:

```
print("%d %d %d %d\n", sizeof(char), sizeof(int),
                       sizeof(long), sizeof(double));
// output: 1, 4, 4, 8
```

Includendo `<limits.h>` si possono ottenere limiti minimi e massimi per ciascun tipo.

```
print("%d %d %u\n", INT_MAX, UINT_MAX, UINT_MAX);
// output: 2147483647 -1 4294967295
```

La funzione `printf` verrà trattata nel prossimo paragrafo.

In

```
unsigned x=1000
```

```
if(x>-1) ...
```

nell'`if` -1 viene convertito anch'esso in `unsigned int` perchè `x` è di questo tipo, per cui viene effettuato il confronto `if(1000>4294967295)` che è sempre falso.

2. Tipi enumerativi definiti mediante `enum`

La dichiarazione `enum{alfa=3,beta,gamma,delta=10,epsilon};` definisce delle costanti intere con valori alfa=3, beta=4, gamma=5, delta=10, epsilon=11. Essendo questi valori costanti la dichiarazione può essere scritta in un file `.h` (che verrà incluso con una direttiva `#include`). I tipi enumerativi vengono spesso usati per parametri di ramificazione, ad esempio per porter scegliere una funzione da eseguire.

```
enum {x1,x2,x3,x4,radx,logx,};
double Fun(double x, int k)
  {switch (k) {case x1:return x; case x2:return x*x; case
  x3:return x*x*x; case x4 return x*x*x*x;
  case radx:return sqrt(x); case logx:return log(x);}}
```

La funzione così definita verrà tipicamente chiamata tramite `y=Fun(x,k);`.

Il tipo `void` viene usato soprattutto in due modi: da un lato per indicare funzioni che non restituiscono risultato (come in `void f()`), dall'altro lato per definire puntatori di tipo generico (indirizzi puri) che possono poi essere convertiti in puntatori di tipo determinato a seconda dei casi.

CAPITOLO 4

La funzione `printf`

Abbiamo già usato più volte la funzione `printf` per l'output formattato. Consideriamo un altro esempio:

```
print("%3d %-12.4f\n",m,x);                                    (*)
```

Il primo parametro di `printf` è sempre una stringa. Questa può contenere delle istruzioni formato che iniziano con `%` e indicano la posizione e il formato per la visualizzazione degli argomenti aggiuntivi. In (*) `%3d` tiene il posto per il valore della variabile `m` che varrà visualizzata come intero di tre cifre, mentre `%-12.4f` indica una variabile di tipo `double` di al massimo dodici caratteri complessivi (compreso il punto decimale) di cui quattro cifre dopo il punto decimale allineati a sinistra a causa di `-` (l'allineamento di default avviene a destra). I formati più usati sono:

`%d`	intero
`%ld`	intero lungo
`%n`	intero ≥ 0
`%ln`	intero lungo ≥ 0
`%c`	carattere
`%f`	double
`%s`	stringa
`%x`	rappresentazione esadecimale
`%o`	rappresentazione ottale

Per ottenere un segno di `%` si usa `%%`. All'interno della specificazione del formato si possono usare `-` per l'allineamento a sinistra, `0` per indicare che al posto di <space> venga usato `0` come carattere di riempimento negli allineamenti a destra. Come visto, è anche possibile precisare il numero di caratteri da usare; con `*` questo numero diventa esso stesso variabile e viene indicato negli argomenti aggiuntivi.

```
int x=441;
for (k=0;k<7;k++) printf("%0*d\n",k,x);

// output: 441
           441
           441
           441
           0441
           00441
           000441
           0000441

double x=1288,3456; int i=9;
```

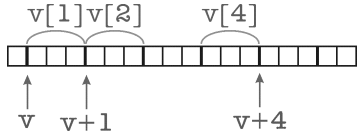
```
for (k=0;k<6;k++) printf("%0*.*f\n",i,k,n);

// output: 000001288
           0001288.3
           001288.35
           01288.346
           1288.3456
           1288.34560
```

Esiste anche una funzione di input formattato `scanf` un po' difficile da usare.

CAPITOLO 5

Vettori e puntatori

Il C colloca gli elementi di un vettore in posizioni di memoria adiacenti associando l'indirizzo più basso al primo elemento e il più alto all'ultimo.



Il formato generale della dichiarazione di un vettore a una dimensione è il seguente: `tipo v[n];`

`n` deve essere di tipo `int` o compatibile con `int`. Il k-esimo e elemento è `v[k]`. Gli indici vengono contati a partire da 0; gli elementi del vettore dichiarato vanno quindi da `v[0]` a `v[n-1]`. L'indicazione della lunghezza `n` ha il solo effetto di riservare la memoria necessaria per il vettore (nel nostro caso n*sizeof(tipo) byte). Se `v` viene usato in sola lettura gli si può assegnare anche una lunghezza indeterminata: `tipo v[];`

Esempio: Programma che calcola la media.

```
double Media(double x[], int n)
 {double s; int k;
  for (s=0,h=0;k<n;k++) s+=x[k];
  return s/n;}
```

Nell'esempio appena visto abbiamo usato l'operatore abbreviato `+=` infatti invece di `a=a+b` si può anche scrivere `a+=b` e similmente si possono abbreviare assegnazioni che usano altri operatori binari:
`a-=b` al posto di `a=a-b`,
`a*=b` al posto di `a=a*b`,
`a/=b` al posto di `a=a/b`,

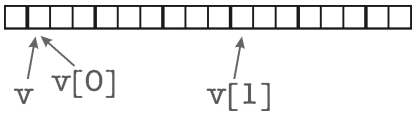
Questa è una notazione buona e comoda; ad esempio: `resistenza/=2;` è più leggibile di `resistenza = resistenza/2;`

Nell'esempio precedente abbiamo anche usato l'operatore `++`. `k++` corrisponde a `k=k+1;`. `k` deve essere di tipo intero. Esiste anche `++k` con la seguente differenza quando queste istruzioni vengono usate all'interno di altre espressioni:

in `f(x++);` viene prima eseguita `f(x)` e successivamente aumentata la `x`. In f(++x); il valore di `x` viene aumentato di uno prima dell'esecuzione di `f`. Nello stesso modo si usa `k--` per `k=k-1`.

Vettori a due dimensioni hanno la dichiarazione: `tipo v[n][m];`. `v[i][j]` sarà poi l'elemento $v_{i,j}$ del vettore; naturalmente anche qui si comincia a contare da 0. Si opera concretamente in memoria quindi `v[0], ..., v[n-1]` sono a loro volta dei vettori ad `n` componenti.

Ad esempio, dopo `int v[2][3];` abbiamo:



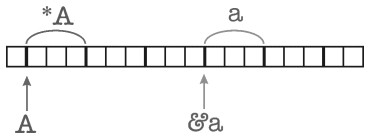
`v` e `v[0]` puntano allo stesso indirizzo, ma `v` è di tipo `int*` che vedremo in seguito e `v[0]` di tipo `int`.

Un vettore a quattro dimensioni può essere dichiarato con tipo `v[n1][n2][n3][n4];`. Se un vettore ha più dimensioni, soltanto la prima dimensione può essere indeterminata ossia tipo `v[][50];`.

Una variabile che viene utilizzata per indicare un indirizzo tipizzato in memoria è un puntatore. In altre parole un vettore è un indirizzo fisso insieme ad un tipo; un puntatore è un indirizzo variabile insieme ad un tipo.

Puntatori vengono usati in due modi: da un lato possono essere utilizzati come indirizzi variabili per muoversi all'interno della memoria, dall'altro vengono usati in modo simile ai vettori per destinare delle aree di memoria in cui conservare dei dati. In tal caso bisogna riservare quest'area di memoria con una delle istruzioni di allocazione di memoria che impareremo più avanti oppure con l'assegnazione diretta di una stringa. Per distinguere puntatori da vettori useremo iniziali maiuscole per i puntatori e in genere minuscole per i vettori. Questa è solo un'abitudine che si propone, ma non è necessaria.

La dichiarazione di un puntatore ha questo formato: `tipo *A;`. Se `A` è un puntatore e `a` una variabile gli operatori `*` e `&` hanno il seguente significato: `*A` è il contenuto a cui punta `A` e `&a` è il puntatore ad `a`.



Le istruzioni `a=7;` e `*&a=7;` hanno lo stesso effetto. I puntatori sono variabili come le altre e quindi possono essere modificati: ad esempio con

```
int[]={1,3,8,7}, *X;
 X=a+2;
```

si definiscono un vettore `a` di interi e un puntatore `X` a interi che successivamente punta ad `a[2]`, cioè in questo caso al valore 8.

Non sono invece permesse le istruzioni `a=a+2;` oppure `a=x;` poichè a è un vettore fisso e quindi non posso muoverlo. A parte questo, molte operazioni sono uguali per puntatori e vettori. Ad esempio, dopo

```
int *X,a[10];
 X=a;
```

Dopo questo le istruzioni `a[4]=7;` e `x[4]=7;` sono equivalenti.

1. Aritmetica dei puntatori

Puntatori vengono spesso usati come variabili in cicli come nel caso:
`for(X=a;X-a<4;X++) printf`(Se `X` è un puntatore (o vettore) di tipo `t` e `n` un'espressione di tipo intero (o intero lungo), allora `X+n` è il puntatore dell stesso tipo `t` e indirizzo uguale a quello di `X` aumentato di `nd` dove `d` è lo spazio che occupa un elemento del tipo `t`. Quindi se immaginiamo la parte di memoria che inizia nell'indirizzo corrispondente ad `X` occupata da elementi

di tipo `t`, `X+n` punta all'elemento con indice `n` (cominciando sempre a contare da 0). In altre parole `*(X+n)` equivale a `X[n]`.

Puntatori possono essere confrontati fra di loro (hanno senso quindi le espressioni `X<Y` oppure `X==Y` per due puntatori `X` ed `Y`); si possono inoltre usare le istruzioni `X++` e `X--` come per variabili intere.

L'essenziale equivalenza tra puntatori e vettori implica che dopo

```
tipo a[100], *A;
A=a;
```

le espressioni `a[k]`, `A[k]`, `*(A+k)`, `*(a+k)` indichino lo stesso elemento. La differenza tra vettori e puntatori consiste solo nel fatto che il puntatore è variabile, mentre il vettore corrisponde ad una posizione fissa in memoria.

Se `A` e `B` sono puntatori dello stesso tipo allora `B-A` è il numero degli oggetti di questo tipo tra `A` e `B` (`*A` e `*B` compresi).

In altre parole `B-A=n` se e solo se `B=A+n`.

2. Passaggio di parametri

Consideriamo la seguente funzione che dovrebbe aumentare di uno il valore di un numero intero.

```
void aumenta(int x)
 {x++;}
```

Facciamo una prova:

```
void prova()
 {int x=5;
  aumenta(x); printf("%d\n";x);}
```

Viene stampato 5, quindi non funziona. I parametri (argomenti) di una funzione in C vengono sempre passati *per valore*. Con ciò si intende che in una chiamata `f(x)` alla funzione `f` viene passato solo il valore di `x` con cui la funzione esegue le operazioni richieste, ma senza che il valore della variabile `x` venga modificato. Anche nel caso che all'interno della funzione ci sia un'istruzione del tipo `x=nuovo valore;` infatti la variabile `x` che appare all'interno della funzione è un'altra variabile che riceve come valore iniziale il valore della `x`. Per aumentare `x` dobbiamo perciò accedere al suo indirizzo. La funzione corretta è quindi:

```
void aumenta(int *A)
{(*A)++;}
```

Oppure naturalmente anche

```
void aumenta(int *A)
{*A=*A+1;}
```

Quando si applica la funzione bisogna usare `&x`:

```
void prova()
 {int x=5;
 aumenta(&x); printf(%d\n",x);}
```

Nel primo caso della funzione `aumenta` non si può usare `*A++` poichè ciò aumenterebbe l'indirizzo `A` (secondo le regole dell'aritmetica dei puntatori) senza alcun altro effetto.

Per la stessa ragione è corretta la funzione:

```
void stampatutti(int*A)
 {for(;*A>=0;A++) printf("%d ",*A);
 print("\n");}
```

che a partire da quell'indirizzo stampa tutti gli interi che incontra fino a quando incontra un intero negativo.

Quando la funzione viene utilizzata, non è il puntatore `A` che si muove, ma una copia locale creata per la funzione. Quindi, dopo l'esecuzione della funzione `A` punta ancora all'inizio del vettore e non al primo intero negativo incontrato. Perciò

```
void prova()
 {int a[]={3,5,8,0,4,-1};
 stampatutti(a); stampatutti(a);}
```

stampa correttamente per due volte la stessa serie di numeri.

3. Puntatori generici

Talvolta il programmatore avrebbe bisogno di strutture e operazioni che funzionino con elementi di tipo qualsiasi. Allora si possono usare puntatori generici che formalmente vengono dichiarati come `void*`. Un esempio:

```
void applica(void(*f)(),void*X)
 {f(X);}

void scrivi(int *X)
 {printf("%d\n", *X);}

void aumenta(int *X)
 {(*X)++;}
```

Adesso con:

```
int a=8; applica(aumenta,&a);
applica(scrivi, &a);
```

otteniamo come output `9`. Si osservi il modo in cui la funzione viene dichiarata come argomento di un'altra funzione.

Un puntatore `void*` può essere considerato come un indirizzo puro.

4. Conversioni di tipo

Puntatori di tipo diverso possono essere convertiti tra di loro. Se `X` è un puntatore di tipo `t` e `s` è un altro tipo, allora `(s)X` è il puntatore con lo stesso indirizzo di `X`, ma di tipo `s`. Ad esempio `X+2` punta all'elemento di tipo `t` con indice 2 a partire dall'indirizzo corrispondente ad `X`, ma `(char*)X+2` punta al secondo byte a partire da quell'indirizzo. Qual è invece il significato di `(char*)(X+2)`?

Conversioni di tipo fra puntatori sono frequenti soprattutto quando si utilizzano puntatori generici (indirizzi puri).

```
void *A; int *B;
B=(int*)A;
```

In questo modo il puntatore `B` di tipo `int` punta sull'indirizzo corrispondente ad `A`.

5. Allocazione di memoria

Per riservare o liberare parti di memoria in C si usano quattro funzioni i cui prototipi sono:

```
void *malloc(size_t n);
void *calloc(size_t n, size_t dim);
void *realloc(void *A, size_t n);
void free(void *A);
```

Tutte queste funzioni richiedono gli header `<stdlib.h>` e vengono usate nel modo seguente.

```
A=malloc(n);
```

Questa istruzione chiede al sistema di cercare in memoria uno spazio di `n` byte attigui all'inizio del quale punterà `A`; se ciò non è possibile `A` viene posto uguale a zero. Per `n` uguale a 0 si ottiene spesso, ma non necessariamente, il puntatore nullo. Per controllare il buon esito dell'operazione in genere l'istruzione sarà seguita da `if(!A) eccezione`.

La funzione `calloc` è un caso particolare di `malloc` infatti: `A=calloc(n,dim)` è equivalente a `A=malloc(n*dim)` e riserva quindi lo spazio per `n` oggetti di `dim` byte ciascuno. Se necessario la dimensione può essere calcolata mediante `sizeof` come in questo esempio: `A=calloc(100, sizeof(unvettore));`.

`realloc` viene usato per modificare lo spazio precedentemente riservato con `malloc` `calloc` o un altro `realloc`. Le regole più importanti per l'uso di realloc sono:

1. `A=realloc(0,n);` è equivalente ad `A=malloc(n);`
2. `A=realloc(A,0);` con $A \neq 0$ equivale essenzialmente a `free(A); A=0;`
3. `A=realloc(A,n);` con `n` non maggiore dello spazio già riservato per `A` libera lo spazio non più richiesto e non modifica l'indirizzo a cui punta `A`. Altrimenti viene riservato più spazio a partire da A se ciò è possibile oppure, in caso contrario, questo spazio viene cercato in un'altra parte della memoria.
4. free(A) fa in modo che lo spazio riservato per `A` venga liberato.
   ATTENZIONE: Se lo spazio in `A` non è riservato (ad esempio a causa di una chiamata di troppo di `free` o perchè non è mai stato allocato) ciò provoca quasi sempre un brutto errore in memoria (*segmentation fault*) tranne nel caso che `A==0`; quindi per sicurezza conviene fare `free(A); A=0;`.

Come vedremo fra poco il C non prevede un apposito tipo; considerà però in molte funzioni il carattere ASCII 0 come terminatore di stringhe. Quindi le affermazioni del prossimo paragrafo talvolta risultano utili anche per operazioni su testi.

6. Operazioni sui byte in memoria

Le seguenti funzioni sono molto simili alle funzioni per le stringhe che vedremo più avanti e si distinguono da esse per il fatto che il carattere ASCII 0 non ha più un significato speciale; sono dichiarate in `<string.h>`.

```
void*memchr(const void*A, int x, size_t n);
```

`memchr(A,x,n);` restituisce un puntatore al primo `x` tra i primi `n` caratteri a partire da `A` oppure il puntatore nullo (*NULL*) se tra questi caratteri nessuno è uguale ad `x`.

```
void*memset(void*A, int x, size_t n);
```

`memset(A,x,n);` pone i primi `n` caratteri a partire da `A` uguali ad `x` (questo argomento dovrebbe del tipo `unsigned char` anche se nella dichiarazione appare come `int`. Ad esempio `memset(A,0,50)` pone 50 caratteri a partire da `A` uguali a zero. Il risultato è uguale ad `A`, ma normalmente è superfluo. Lo spazio necessario deve essere stato riservato in precedenza.

```
void*memcpy(void*A, const void*B, size_t n);
void*memmove(void*A, const void*B, size_t n);
```

`memcpy(A,B,n)` e `memmove(A,B,n)` copiano entrambe `n` byte da `B` in `A`, ma mentre `memcpy` non può essere utilizzata quando le due regioni di memoria si sovrappongono, `memmove` funziona anche in questo caso. Quindi l'istruzione `memmove(A,B,strlen(B)+1);` può essere usata per copiare la stringa `B` in `A` (compreso il carattere 0 finale) anche nel caso di sovrapposizione in memoria.

Verifichiamo infine il diverso significato di `A+k` per un puntatore a seconda del tipo a cui punta `A`.

```
void Prova()
{char*A; int*B; double*C; int k;
for(k=0;k<3;k++)
printf("%ld %ld %ld\n", A+k, B+k, C+k);}

// output: 1108517584 1108519512 134513422
           1108517585 1108519516 134513430
           1108517586 1108519520 134513438
```

CAPITOLO 6

Stringhe

1. Introduzione

Il C non possiede un tipo stringa. Le stringhe vengono invece rappresentate con vettori di caratteri. Ciò non è uno svantaggio; ne consegue anzi una maggiore flessibilità e potenza nel trattamento delle stringhe rispetto a molti linguaggi che prevedono un tipo di stringa apposito con operatività limitata. Una stringa in C è semplicemente una successione di caratteri terminata dal carattere con codice ASCII zero. Nella rappresentazione esplicita stringhe vanno racchiuse tra virgolette `"..."` caratteri tra apostrofi `'...'`.

2. Alcuni esempi

Stringhe possono essere inizializzate in questo modo abbreviato:

```
char a[]="Pentecoste";
char *A="Pasqua";
```

Un vettore di stringhe viene rappresentato da un vettore a due dimensioni:

```
char a[10][20];
char **A;
```

con possibili inizializzazioni:

```
char a[2][20]={"Dante","Petrarca"};
char *A[2]={"Dante","Petrarca"};
```

Per capire queste costruzioni bisogna pensare sempre in termini di memoria. In parte le dichiarazioni `char *A;` e `char a[100];` si equivalgono. Esiste però una differenza (che vale per ogni tipo di puntatore): se si usa un puntatore con `char *A;` non viene riservato lo spazio per una stringa; se si usa un vettore (con indicazione della lunghezza) lo spazio richiesto viene invece assegnato e la variabile del vettore contiene l'indirizzo dell'inizio di quel tratto di memoria riservato. Mentre per altri tipi di vettori, ad esempio di interi, è permessa una inizializzazione, ad esempio `int a[]={3,5,8,0,4,-1};` l'inizializzazione analoga per puntatori è possibile solo per puntatori a caratteri (cioè stringhe). Quindi mentre non possiamo scrivere `int *A={3,5,8,0, 4,-1};` l'istruzione `char *A="Alberto";` è permessa oppure anche un'assegnazione successiva come `char *A; A="Alberto";`. In questo caso viene cercato prima uno spazio di 8 byte adiacenti che viene riempito con i caratteri `'A' 'l' ... 'o' 0`, dopo di che A diventa l'indirizzo del primo byte della stringa. Lo spazio riservato per `A` è, in questo caso, di esattamente 8 byte; È quindi possibile trasformare la stringa in `Roberto` mediante le istruzioni `A[0]='R'; A[1]='o';` mentre `A[20]=0;` causerà quasi sicuramente un errore poiché si scriverà su uno spazio non più riservato. Se invece scriviamo `A="Robero";` vengono cercati altri 8 byte liberi in memoria al primo dei quali

punterà adesso `A`. Ciò invece non è possibile dopo char `a[]="Alberto";`, In questo caso a è un indirizzo fisso che non può essere spostato.

3. Virgolette e apostrofi

A differenza delle stringhe, singoli caratteri vengono racchiusi tra apostrofi `'`: ad esempio

```
char a, b, c;
a='x'; b='y'; c='z';
printf("%c %c %c\n",a,b,c);
// output: x y z
```

La seguente istruzione scrive il codice ASCII sullo schermo:

```
int n;
for(n=0;n<256;n++)
printf("%3d %c\n",n,n);
```

La differenza tra `'A'` e `"A"` è che `'A'` è un carattere singolo, mentre `"A"` in memoria è rappresentato da `A \0` (in quanto nelle stringhe è sempre compreso il carattere finale).

Per inserire `"` si usa `\"` come nel seguente esempio:
`printf ("Disse \"Ciao!\", sorridendo.\n");` che dà come output:
`Disse "Ciao!", sorridendo.`

L'istruzione `char *A; A="Rino";` crea uno spazio riservato di 5 byte in una determinata posizione di memoria. Non dobbiamo perciò scrivere in `A+5` con `A[5]='x';` perchè così si provocherà un errore di memoria (come anche l'istruzione `A=A+50; *A='x';` se lo spazio non è stato riservato in altro modo in precedenza). È invece corretto `A="Venezia";` perchè in tal caso viene creato un nuovo spazio in memoria, lasciando però riservato con il contenuto precendente di `A`.

Controlliamo se effettivamente lo spazio originale rimane riservato:

```
void Prova()
{char *A, *B; int k;
A="Rino"; B=A; A="Venezia";
for(k=0;k<5;k++) *B='M';
printf("%3 %5\n",B,A);}

// output: Mino Venezia
```

4. Alcune funzioni

La lunghezza di una stringa `A` viene calcolata con `strlen(A)`. Possiamo facilmente creare una funzione nostra allo stesso scopo.

```
int Lun(char *A)
 {int n;
 for(n=0;*A;A++,n+1); return n;}
```

Abbiamo già osservato in precedenza che la funzione non modifica l'indirizzo `A` perchè internamente lavora con una copia. La costruzione `for(n=0; *A; A++,n+1); return n; ...` è tipicamente usata per percorrere una stringa.

Confronto di stringhe. Definiamo la funzione Us per l'uguaglianza di stringhe nel modo seguente:

```
int Us(char *A, char *B)
 {for(;*A;A++,B++) if(*A!=*B) return 0;
 return(*B==0);}
```

L'algoritmo percorre la prima stringa fino alla sua fine e confronta ogni volta il carattere nella prima stringa con il carattere nella posizione corrispondente della seconda. Quando trova la fine della prima controlla ancora se anche la seconda termina. Si noti che anche qui per percorrere le due stringhe usiamo le stesse variabili `A` e `B` che all'iniziano denotano gli indirizzi delle stringhe. Abbiamo infatti già osservato che questo non cambia i valori originali di `A` e `B`, ma solo le loro copie interne nella funzione.

Us(A,B) restituisce il valore 1 se le due stringhe sono uguali, altrimenti 0. Per provarlo possiamo usare la funzione:

```
void Prova()
 {print("%d %d %d\n",
 Us("alfa","alfabeto"),
 Us("alfa","alfa"),
 Us("alfa","beta"));}
// Output: 0 1 0
```

In verità per il confronto di stringhe conviene usare la funzione `strcmp` del C di cui parleremo tra poco.

```
int Us(char *A, char *B)
 {return (strcmp(A,B)==0);}
```

Ricordiamo che l'espressione booleana in C è un numero (uguale a 0 o a 1) che può essere risultato di una funzione.

Attenzione: Per l'uguaglianza di stringhe non possiamo usare `(A==B)` perchè riguarderebbe l'uguaglianza degli indirizzi in cui si trovano le due stringhe (una condizione, quindi, molto più forte).

Definiamo adesso una funzione `Uis` (=Uguaglianza inizio stringhe) di due stringhe che restituisce 1 o 0 a seconda che la prima stringa è sottostringa della seconda oppure no.

```
int Uis(char *A, char *B)
 {for(;*A;A++,B++) if(*A!=*B) return 0;
 return 1;}
```

`Us` e `Uis` si distinguono solo nell'ultima riga. Anche in questo caso potremmo usare una funzione della libreria standard:

```
int Uis(char *A, char *B)
 {return (strncmp(A,B,strlen(A))==0);}
```

Per portare un puntatore `X` alla fine di una stringa `A` in modo che `X` punti sul carattere `'\0'` finale si può usare l'istruzione: `for(X=A;*X;X++);` oppure usando la funzione di libreria `X=strchr(A,0);`. Anche su questa funzione torneremo tra poco.

5. Input da tastiera

Per l'input da tastiera, in casi semplici si può usare la funzione `gets`:

```
char a[40];
gets(a);
```

Il compilatore può avvertire della pericolosità di questa funzione con il messaggio: `The 'gets' function is dangerous and should not be used`. Infatti, se l'utente immette più di 40 caratteri (per disattenzione o per danneggiare volutamente il sistema) scriverà su posizioni non riservate della memoria. Nei nostri esperimenti ciò non è un problema, ma può essere importante in programmi che verranno usati da utenti poco esperti oppure malintenzionati.

Si preferisce per questa ragione la funzione `fgets`:

```
char a[40];
fgets(a,38,stdin);
```

In questo caso nell'indirizzo `a` vengono scritti al massimo 38 caratteri. Ricordiamo che `stdin` è lo *standard input* cioè normalmente la tastiera (`fgets` può ricevere il suo input anche da altri files).

A differenza di `gets`, `fgets` inserisce nella stringa anche il carattere `'\n'` e ciò è un po' scomodo. Definiamo quindi una nostra funzione di input:

```
void Input(char *A, int n)
 {if(n<1) n=1;
 fgets(A,n+1,stdin);
 for(;*A;A++);A--;
 if(*A=='\n')*A=0;}
```

Usando la funzione `strchr` del paragrafo precedente possiamo riscrivere `Input` nel modo seguente:

```
void Input(char *A, int n)
 {if(n<1) n=1;
 fgets(A,n+1,stdin);
 A=stdchr(A,0)-1;
 if(*A=='\n')*A=0;}
```

Esempio: il fattoriale. Il fattoriale n! di un numero naturale può essere calcolato con un algoritmo iterativo:

```
double Fattoriale(int n)
 {double f; int k;
 for(f=1,k=1;k<=n;k++) f*=k;
 return f;}
```

oppure con un algoritmo ricorsivo:

```
double Fattoriale(int n)
 {if(n<=1) return 1;
 return n*Fattoriale(n-1);}
```

6. Funzioni utili per l'input

Nelle operazioni di input si usano spesso le funzioni `atoi`, `atol` e `atof` che convertono una stringa in un numero (rispettivamente di tipo `int`, `long` e `double`). Bisogna ricordarsi di includere `<stdlib.h>`. La seguente funzione chiede ripetutamente l'input di `n` e stampa `n!` terminando quando $n \leq 0$:

```
void Prova()
 {char a[40]; int n;
 while(1) {printf("n: "); Input(a,35); n=atoi(a); if(n<=0) break;
 printf("%.0f\n", Fattoriale(n));}}
```

Si tenga conto che per $n \geq 23$ (ad esempio) le cifre previste per il formato `%f` non sono più sufficienti. Anche con il tipo double il C non riesce più a calcolare correttamente con numeri così grandi. Si possono usare il programma `calc` oppure in C la libreria `gmp`.

Esempio: il coefficiente binomiale. Calcoliamo i numeri binomiali usando la formula:

$$\binom{n}{k} = n\ (n-1)\ ...\ (n-k+1)$$

Anche qui i risultati saranno corretti per numeri relativamente piccoli.

```
double Bin(int n, int k)
 {double num, den, i, j;
 for(num=den=1,i=n,j=1;j<=k;i--,j++)
 {num*=i;den*=j;}
 return num/den;}
```

modificando la funzione `Prova` nel modo seguente:

```
void Prova()
 {char a[40]; int n, k;
 while(1) {printf("n: "); Input(a,35);
 n=atoi(a); if(n<=0) break;
 printf("k: "); Input(a,35);
 k=atoi(a); if(n<=0) break;
 printf("%.0f\n", Bin(n,k));}}
```

Esempio: invertire le parole. La seguente funzione chiede (ripetutamente fino a quando non immettiamo la parola vuota) una parola, conferma la parola impostata e visualizza la parola che si ottiene dall'originale invertendone la successione dei caratteri e usando solo lettere minuscole.

```
void Invertiparola()
 {char parola[100], inversa[100], *X, *Y;
 while(1) {printf("\nQuale parola vuoi invertire? ");
 Input(parola,70); if(Us(parola,"")) break;
 printf("La parola originale e' %s.\n",parola);
 for(X=strchr(parola,0)-1,Y=inversa; X>=parola; X--, Y++)*Y=*X;
 *Y=0;
 for (Y=inverse; *Y,Y++) *Y=tolower(*Y);
 printf("invertita diventa: %s.\n",inversa);}}
```

Si osservi che nel primo `for X` viene inizialmente posto a puntare sull'ultimo carattere della parola con `X=strchr(parola,0)-1;`. La penultima riga trasforma tutte le lettere della stringa invertita in minuscole utilizzando la funzione `tolower` che richiede il header `<ctype.y>` come la sua gemella `toupper` che converte un carattere in maiuscola.

Esempio: eliminare caratteri. Definiamo adesso una funzione che permette di inserire una stringa da cui verranno eliminati tutti i caratteri che appaiono in una seconda stringa anch'essa impostata dalla tastiera.

```
void Eliminacaratteri()
 {char a[100],b[100],*X,*Y,*C; int del;
 printf("\nInserisci parola: "); Input(a,80);
 printf("\nInserisci i caratteri da eliminare: "); Input(b,40);
 for(X=Y=a;*X;X++)
 {for(C=b,del=0;*C;C++)
 if(*X==*C){el=1; break;}
 if(!del) *(Y++)=*X;} *Y=0;
 printf("\n%s\n",a);}
```

Non dimenticare di chiudere la stringa ridotta con `*Y=0;`.

Potremmo semplificare questa funzione utilizzando la libreria per le stringhe del C, come vedremo in seguito.

La funzione `isdigit`. La funzione `int isdigit(int x);` dovrebbe essere usata per $0 \leq x \leq 127$ oppure `x==EOF` (= *End Of File*) e restituisce 1 se $x \in \{'0','1',...,'9'\}$. Può essere usata per controllare se una stringa consiste solo di cifre per poter essere considerata come numero naturale (zeri iniziali superflui sono permessi) come nella seguente funzione:

```
int naturale(char *A)
 {for(;*A;A++) if(!isdigit(*A)) return 0;
 return 1;}
```

CAPITOLO 7

La funzione `sprintf`

1. Introduzione

La funzione `sprintf` è molto simile nella sintassi alla funzione `printf` da cui si distingue per un argomento in più che precede i tipici argomenti di `printf`. Il primo argomento è un puntatore a caratteri e la chiamata della funzione fa in modo che i caratteri, che con `printf` verrebbero scritti sullo schermo, vengano invece scritti nell'indirizzo che corrisponde a quel puntatore.

Ci sono due usi principali di questa funzione: da un lato può essere utilizzata per creare delle coppie di stringhe, dall'altro può servire per preparare una stringa per una successiva elaborazione ad esempio per un output grafico che non utilizza `printf`.

```
char a[200];
 sprintf(a,"Il valore e' %.2f", x);
 scrivinellafinestra(f,a);
```

dove immaginiamo che l'ultima istruzione effettua una visualizzazione della stringa `a` nella finestra `f`

2. Copiare stringhe

La funzione `sprintf` può essere usata per copiare una stringa, bisogna però stare attenti a due cose: in primo luogo la parola da copiare non deve contenere caratteri che possono essere interpretati come caratteri di formattazione (come ad esempio \ e %) inoltre la copia deve essere disgiunta in memoria dall'originale. Non funziona perciò la seguente istruzione:

```
char a[200]="Bologna";
 printf("%s\n",a);
 sprintf(a+2,a);
 printf("%s\n",a);
```

La seconda riga dell'output inizierà con BoBoBo... e talvolta si avrà un `segmentation fault` indice di sovrascrittura di memoria non riservata.

Nello stesso modo si comporta la seguente funzione:

```
void copianonsicura(char *A,char *B)
 //le due stringhe non devono
 //sovrapporsi in memoria
 {for(;*A;A++,B++)*B=*A;*B=0;}
```

Funziona invece anche nel caso di sovrapposizione in memoria:

```
void copia(char *A,char *B)
 {char *X,*Y,*Z;
 X=malloc(strlen(A)+4;
```

```
for(Y=X;*A;A++,Y++)*Y=*A;*Y=0;
for(Y=B,Z=Y;*Z;Z++,Y++)*Y=*Z;
*Y; free(X);}
```

Qui abbiamo usato le due funzioni `malloc` e `free` descritte in precedenza. Il vantaggio di `sprintf` è comunque che permette l'inserimento di parti variabili nella stringa da copiare.

Un altro modo per effettuare una copia sicura di una stringa `B` in una stringa `A` è l'uso dell'istruzione `memmove(A,B,strlen(B)+1);` che usa la funzione `memmove` anch'essa già vista in precedenza.

CAPITOLO 8

Le funzioni per le stringhe del C

1. Introduzione

Le librerie standard del C prevedono numerose funzioni per il trattamento di stringhe che bisognerebbe conoscere. Spesso non sarebbe difficile creare funzioni apposite simili, ma le funzioni standard sono ottimizzate e se si conosce il loro funzionamento affidabili. Esse richiedono gli header `<string.h>`.

Le più importanti, tra queste funzioni, sono:

1. una funzione per calcolare la lunghezza di una stringa: `strlen`;
2. due funzioni per il concatenamento di stringhe: `strcat` e `strncat`;
3. due funzioni per la copia di stringhe: `strcpy` e `strncpy`;
4. due funzioni per il confronto di stringhe: `strcmp` e `strncmp`;
5. due funzioni per la ricerca di un singolo carattere in una stringa: `strchr` e `strrchr`;
6. tre funzioni per la ricerca in una stringa di un carattere contenuto in un insieme di caratteri: `strspm`, `strcspm` e `strpbrk`;
7. una funzione per la ricerca di una stringa in una stringa: `strstr`;
8. una funzione per la separazione di una stringa in sottostringhe delimitate da separatori: `strtok`;

2. Esempi introduttivi

```
size_t strlen(const char *A)
```

Questa funzione restituisce il numero di caratteri della stringa `A` senza contare il carattere ASCII zero. La stringa vuota ha la lunghezza pari a zero. Abbiamo già incontrato in precedenza questa funzione nei capitoli 5, 6 e 7. Un altro esempio è:

```
printf("%d %d \n", strlen(" "), strlen("John\n Bob\n"));
// output: 0 9
```

Per il concatenamento di stringhe si possono usare le funzioni

```
char *strcat(char *A, const char *B);
char *strncat(char *A, const char *B, size_t n);
```

Attenzione però poichè l'istruzione `strcat(A,B);` fa in modo che `A` diventi uguale alla concatenazione di `A` e di `B`. In altre parole il carattere zero finale di `A` viene sostituito con il primo carattere di `B`. Bisogna stare attenti che per `A` sia riservato sufficiente spazio. In particolare è un grave errore l'istruzione `X=strcat("alfa","beta");`.

Esempio di uso corretto:

```
char a[100]="alfa";
strcat(a,"beta");
```

```
printf("%s\n",a);
// output: alfabeta
```

Le funzioni restituiscono come risultato il primo argomento.

Attenzione: le stringhe `A` e `B` non si devono sovrapporre in memoria. L'istruzione `strcat(A,A+4);` sarà quasi certamente un errore se `A` consiste in più di tre caratteri.

`strncat(A,B,n);` funziona nello stesso modo (e richiede le stesse precauzioni), ma aggiunge solo i primi n caratteri di `B` ad `A`, mettendo quando necessario un carattere zero alla fine della nuova stringa:

```
char a[100]="alfa";
strncat(a,"012345",3);
printf("%s\n",a);
//output: alfa012
```

Le istruzioni `strcat(A,B);` `strncat(A,B,strlen(B));` e `strncat(A,B,n);` con $n \geq$`strlen(B)` sono equivalenti.

3. strcpy e strncopy

Prototipo:

```
char *strcpy(char *A, const char *B);
char *strncpy(char *A, const char *B, size_t n);
```

`strcpy(A,B);` copia `B` in `A`, `strncpy(A,B,n);` copia i primi `n` caratteri di `B` in `A`. Se la lunghezza di `B` è minore di `n`, i caratteri mancanti vengono considerati uguali a zero. Se invece n è minore o uguale della lunghezza di `B`, allora non viene trasferito uno zero e la nuova fine di `A` è il primo zero che si incontra a partire da `A`.

Le funzioni restituiscono come risultato (superfluo) il primo argomento.

Esempi:

```
char[100]="01234";
strcpy(a,"abc"); puts(a);     // output: abc
strcpy(a,"012345"); puts(a);  // output: 012345
strcpy(a,"abcde"); puts(a);   // output: abc345
strcpy(a,"012345"); puts(a);  // output: 012345
strncpy(a,"abc",7); puts(a);  // output: abc
strcpy(a,"012"); strcpy(a+4,"456789"); puts(a);
                              // output: 012
strncpy(a,"abcdefg",5); puts(a);
                              // output: abcde56789
strcpy(a,"abcde"); puts(a);   // output: abcde
```

Abbiamo usato l'istruzione `puts(a);` che è equivalente a `printf("%s\n", a);`. Anche in questo caso bisogna riservare spazio sufficiente per `A`. L'uso di queste funzioni costituisce un errore se `A` e `B` si sovrappongono in memoria. Useremo quindi `memmove` in tal caso.

4. strcmp e strncmp

Le funzioni per il confronto di stringhe `strcmp` e `strncmp` hanno i prototipi:

```
char *strcmp(const char *A, const char *B);
char *strncmp(const char *A, const char *B, size_t n);
```

e le abbiamo già incontrate in precedenza.

`strcmp(A,B);` confronta alfabeticamente `A` e `B` e restituisce un intero maggiore di zero (in genere 1) se `A` è alfabeticamente maggiore di `B`, altrimenti zero, se le due stringhe coincidono e un intero minore di zero (normalmente -1) se `A` è alfabeticamente minore di `B`.

`strncmp(A,B,n);` è uguale a `strcmp(A',B');` dove `n'=min(n,strlen(A), strlen(B))`; con `A'` e `B'` denotiamo le stringhe che constano dei primi `n'` caratteri di `A` e di `B`. Quindi se `A` o `B` hanno meno di `n` caratteri (ad esempio `strlen(A)=n'`$\leq n$ e `strlen(A)`$\leq$`strlen(B)`) allora solo i primi `n'` caratteri vengono considerati.

In particolare vediamo che `A` è inizio di `B` se e solo se `strncmp(A, B, strlen(A))==0`. Si noti che in questo caso il carattere zero finale di `A` non conta più.

5. strchr e strrchr

`strchr` e `strrchr` hanno i prototipi:

```
char *strchr(const char *A, int x);
char *strrchr(const char *A, int x);
```

`ctrchr(A,x)` restituisce il puntatore zero se `x` (considerato come carattere non appare in `A` (cioè nella stringa che corrisponde ad `A`, compreso il carattere zero finale; altrimenti restituisce un puntatore al primo `x` in `A` (quindi al carattere zero finale se `x` è uguale a zero, come abbiamo già visto in precedenza più volte).

`strrchr` è simile, ma cerca a partire dalla fine della stringa `A`. Più precisamente `strrchr` è uguale al puntatore nullo se `x` non appare in `A`, altrimenti è un puntatore all'ultimo `x` in `A`.

Esercizio: Analizzare questa piccola funzione:

```
size_t Pos(char *A, int x)
 {char *P;
 P=strchr(A,x);
 if(!P) return -1; return P-A;}
```

Usando `strchr` per determinare se un carattere appare in una stringa (infatti `strchr(A,x)==NULL` se e solo `x` non appare in `A`) possiamo semplificare la funzione *Eliminacaratteri* del paragrafo 6 nel modo seguente:

```
void Eliminacaratteri()
 {char a[100],b[100],*X,*Y,*C; int del;
 printf("\nInserisci parola: "); Input(a,80);
 printf("\nInserisci i caratteri da eliminare: "); Input(b,40);
 for(X=Y=a;*X;X++)
 if(!strchr(b,*X)) *(Y++)=*X;
 *Y=0;
 printf("\n %s\n",a);}
```

La seguente funzione sostituisce invece ogni carattere da `L` in `A` con `x`:

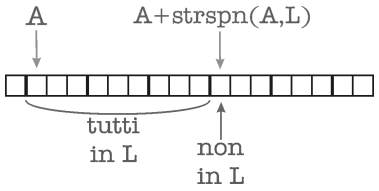
```
void Sost(char *A,char *L, char x)
 {for(;*A;A++) if(strchr(L,*A)) *A=x;}
```

6. `strspn`, `strcspn` e `strpbrk`

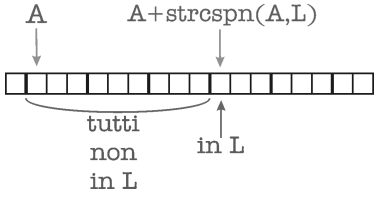
Le funzioni `strspn`[1], `strcspn`[2] e `strpbrk`[3] hanno come prototipi:

```
size_t strspn(const char *A, const char *L);
size_t strcspn(const char *A, const char *L);
char *strpbrk(const char *A, const char *L);
```

`strspn(A,L)` restituisce la lunghezza del segmento iniziale di `A` che consiste di caratteri di `L` (e quindi la lunghezza di `A` se tutti i caratteri di `A` appartengono a `L`).



`strcspn` funziona in modo complementare a `strspn`; più precisamente `strcspn(A,L)` è la lunghezza del segmento iniziale di `A` che non consiste di caratteri di `L` (e quindi la lunghezza di `A` se nessun carattere di `A` appartiene ad `L`).



Esempi:

```
printf("%d\n", strspn("0123456","2015"));
// output: 3

printf("%d\n", strspn("0123456","xy"));
// output: 0

printf("%d\n", strspn("012121","012"));
// output: 6

printf("%d\n", strspn("0123456","04"));
// output: 1

printf("%d\n", strcspn("0123456","2015"));
// output: 0

printf("%d\n", strcspn("0123456","xy"));
// output: 7

printf("%d\n", strcspn("012121","81"));
// output: 1
```

[1] `spn` da *span* = spalmare, estendersi
[2] `cspn` = complemento dello span
[3] `pbrk` = point break

```
printf("%d\n", strcspn("0123456","34"));
// output: 3
```

`strpbrk(A,L)` restituisce il puntatore `NULL` se `A` non contiene caratteri di `L`, altrimenti restituisce un puntatore al primo carattere di `L` in `A`.

`strpbrk(A,"x")` è quindi uguale a `strchr(A,'x')`. Quando `A` contiene caratteri di `L` le istruzioni `X=strpbrk(A,L);` e `X=A+strcspn(A,L);` sono equivalenti. Se invece `A` non contiene caratteri di `L` `strpbrk(A,L)` è il puntatore nullo, mentre `A+strcspn(A,L)` punta al carattere zero finale di `A`.

	$L \cap A \neq \phi$	$L \cap A = \phi$
`strpbrk(A,L)`	comportamento	`NULL`
`X=A+strcspn(A,L)`	equivalente	\0 finale di A

Esperimento:

```
char a[40];
sprintf(a,"012340123pxyz");
a[9]=0;
 //toglie p in a e mette al suo posto 0
X=strpbrk(a,"xyz");
 //dovrebbe dare come risultato il puntatore NULL
Y=a+strcspn(a,"xyz");
if(X) puts(X);
puts(Y+1);
 //output: xyz
```

Viene quindi visualizzato solo `Y` perchè `X` è nullo. Nel seguente caso invece:

```
X=strpbrk(a,"xyz4");
Y=a+strcspn(a,"xyz4");
```

i due valori `X` e `Y` coioncidono.

Lo verifichiamo con:

```
printf("%s\n%s\n%ld\n",X,Y,X-Y);
// output: 40123
           40123
           0
```

`strspn` e `strcspn` vengono spesso usate anche per separare parole in un testo.

Esempio:

```
void prova()
 {char *A,*In,*Fine;
 A="  Romagna \n ";
 In=A+strspn(A," \n");
 Fine=In+strcspn(In," \n"); *Fine=0;
 printf("...%s===\n",In);}
  //output: ...Romagna===
```

7. strstr

La funzione strstr ha il prototipo:

```
char *strstr(const char*A, const char*B)
```

Questa funzione viene utilizzata per cercare una sottostringa in una stringa. `strstr(A,B)` è uguale al puntatore `NULL` se non è sottostringa di `A`, altrimenti è uguale al puntatore alla prima apparizione di `B` in `A`.

7.1. Esempio:

```
char *A="012301850182", *X;
X=strstr(A,"018");
```

```
size_t Posstr(char *A, char*B)
 {char *X;
 X=strstr(A,B); if(!X) return -1; return X-A;}
```

8. strtok

La funzione `strtok` possiede il prototipo:

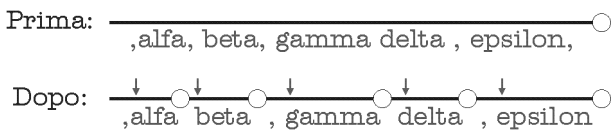
```
char *strtok(char *A, const char*L);
```

`strtok` serve a separare una stringa in sottostringhe, è un po' difficile nell'applicazione e spesso si può usare un metodo apposito più trasparente usando le altre funzioni finora incontrate. Anche qui la stringa `L` serve come contenitore di caratteri. Una caratteristica particolare di `strtok` è che viene chiamata di nuovo per ogni separazione e ogni volta `L` può essere diversa. La prima chiamata è della forma `strtok(A,L1)` (dove `A` è la stringa da separare), le chiamate successive sono della forma `strtok(0,L2)`, `strtok(0,L2)` e così via.

Esempio:

```
char a[]=",alfa, beta, , gamma delta , epsilon,",*A;
 for(A=a;;A=NULL) {A=strtok(A," ,");
 if(!A) break; puts(A);}

 //output: alfa
           beta
           gamma
           delta
           epsilon
```

La stringa che si vuole separare viene *modificata* durante le operazioni! Perciò, se la si vuole utilizzare ancora con il valore originale bisogna copiarla preventivamente. Nel nostro esempio:

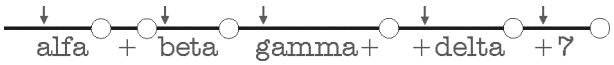


Le frecce indicano le successive posizioni di `A`.

Un esempio di cambio del contenitore `L` nelle chiamate successive è:

```
char a[]="alfa + beta, gamma+, +delta, +7";
A=strtok(a,"+"); puts(A);
for(;;) {A=strtok(0," ,"); if(!A) break; puts(A);}

 //output: alfa
           +
           beta
           gamma+
           +delta
           +7
```

Dopo l'esecuzione in memoria abbiamo la seguente situazione:



`strtok` utilizza un puntatore interno `P` che viene cambiato in ogni chiamata e utilizzato nella chiamata successiva; essa restituisce inoltre ogni volta un altro puntatore `E`. La seguente funzione imita il comportamento di `strtok`:

```
char *strtoknostra(char *A, const char *L)
 {char *E, *Q; static char *P=0;
 E=A? A:P?P:NULL; if(!E) {P=NULL; return NULL;}
 E+=strspn(E,L); if(!*E) {P=NULL; return NULL;}
 Q=strbrk(E,L); if(Q) {*Q=0; P=Q+1;} else P=NULL;
 return E;}
```

La cosa importante è che `P` viene dichiarato di tipo `static`; ciò fa in modo che la funzione si ricordi il valore di `P` nelle varie chiamate come vedremo in seguito. Si vede comunque che spesso sarà meglio programmare appositamente le operazioni, sia per avere un miglior controllo, sia per eventualmente aggiungere altre funzionalità.

`static` e `extern`. Variabili e funzioni dello stesso nome in file diversi devono essere dichiarate `static` altrimenti il *linker* invierà il messaggio `multiple definition of [...]`. Se una variabile dichiarata in un file deve essere invece usata anche da altri file, in questi ultimi (va inserito un *header* comune) deve essere dichiarata di classe `extern` in modo che già all'atto della compilazione il compilatore sappia di che tipo di dati si tratta.

9. Variabili di tipo static all'interno di una funzione

Variabili possono essere dichiarate anche al di fuori di una funzione; in questo caso la classe di memoria `static` significa che la variabile non è visibile al di fuori del file sorgente in cui è contenuta come abbiamo visto nella funzione `Strtoknostra`.

Ciò naturalmente vale ancora di più per una variabile dichiarata internamente ad una funzione. In tal caso l'indicazione della classe `static` ha una conseguenza peculiare che talvolta è utile, ma che può implicare un comportamento della funzione misterioso se non si conosce la regola. Infatti mentre normalmente se una variabile è interna ad una funzione in ogni esecuzione della funzione il sistema cerca di nuovo uno spazio in memoria per questa

variabile alle variabili di classe `static` viene assegnato uno spazio in memoria fisso che rimane sempre lo stesso in tutte le chiamate della funzione (ciò evidentemente ha il vantaggio di impegnare la memoria molto meno); i valori di queste variabili si conservano da una esecuzione all'altra.

Inoltre una eventuale inizializzazione per una variabile `static` viene eseguita solo nella prima chiamata (è soprattutto questo che può confondere).

Esempi:

```
void Provastatic1()
 {static int s=0, k;
 for(k=0;k<5;k++) s+=k; printf("%d\n",s);}
```

Nella prima esecuzione di questa funzione viene visualizzato 10 (la somma dei numeri 0 1 2 3 4), la seconda volta però 20, la terza volta 30 e così via. Questo perchè l'inizializzazione `s=0` viene effettuata solo la prima volta. Probabilmente ciò non è quello che qui il programmatore, che forse intendeva soltanto risparmiare memoria usando una variabile `static`, desiderava fare. Si può comunque rimediare facilmente senza rinunciare a `static` poichè l'istruzione `s=0`, se data al di fuori della dichiarazione, viene eseguita normalmente ogni volta.

```
void Provastatic2()
 {static int s=0, k;
 for(s=k=0;k<5;k++) s+=k; printf("%d\n",s);}
```

Esistono però situazioni in cui può essere utile che una variabile interna di una funzione conservi il suo valore da una esecuzione all'altra. Un esempio importante è la funzione `strtok` che abbiamo imitato con `Strtoknostra` in precedenza. Vedremo adesso un altro esempio tipico.

Un modo appropriato e molto utile dell'uso di variabili `static` interne è la sostituzione di *variabili globali*. Variabili globali, cioè variabili dichiarate al di fuori di una funzione, dovrebbero essere evitate il più possibile. Se una funzione `f` a bisogno di ricordare i valori di una variabile tra le sue successive esecuzioni invece di creare una variabile globale a cui una funzione accede, è di gran lunga preferibile dichiarare una variabile `static` all'interno della `f`. Variabili globali fanno diventare poco trasparenti e difficili da gestire i programmi; in particolare la presenza di funzioni che usano o addirittura modificano il valore di variabili che non appaiono nè tra i loro argomenti, nè tra le variabili interne rendono estremamente intricata la struttura di un programma.

Quando più funzioni devono accedere alla stessa variabile si può dichiarare questa variabile come variabile `static` di una apposita funzione di collegamento che esegue le altre funzioni come vedremo in seguito. Assumiamo di avere una stringa:

in cui ∘ indica il carattere `0` terminale, • il carattere `\n` di nuova riga e `>`

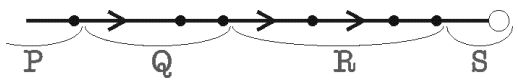


FIGURA 1

un carattere (come ad esempio `>`) che indica che il carattere `\n` successivo

non termina il paragrafo. La stringa è quindi suddivisa in paragrafi P, Q, R, S come indicato in figura 1.

Per stampare i paragrafi usiamo le seguenti funzioni:

```
char *Cerca(char *A, int(*f)())
//cerco 1 posizione nel testo
//che soddisfa una certa condizione
{f(NULL); for(;*A;A++) if(f(A)) return A; return A;}

int Fineriga(char *A)
{static int cont;
if(!A) {cont=0; return -1;}
//normalmente il risultato -1 non serve
//(è solo per correttezza sintattica)
switch(*A) {case '>':cont=1; return 0;
            case 0:return 1;
            case '\n':if(!cont) return 1; cont=0;} return 0;}

void Stampaparagrafi(char *A)
{char *X,*Y,*Fine=ctrchr(A,0);
for(X=A;X<Fine;) {Y=Cerca(X,Fineriga);
*(Y++)=0; printf("%s\n------\n",X); X=Y;}}

void Prova()
{char *A="Il paragrafo > continua\nsu questa seconda riga\n"
"Una riga solo.\nTre > righe \n insieme > \n stavolta\n"
"Un'ultima stringa.";
Stampaparagrafi(A);}
```

Assumiamo di avere tre funzioni `f1`, `f2`, `f3` (per semplicità tutte con un tipo `void`) che utilizzano tutte e tre una stessa variabile ogni volta con il valore che questa variabile ha assunto nella precedente esecuzione di un una delle tre funzioni. Allora possiamo creare una funzione `Gruppo` che coordina questo gruppo di funzioni e al suo interno una variabile `static` come nel seguente esempio:

```
void Gruppo(void (*f)())
{static int a=0; f(&a);}
```

Invece dell'esecuzione diretta `fk()` useremo l'istruzione `Gruppo(fk);`.

Se vogliamo permettere un ulteriore parametro di `f` possiamo definire `Gruppo` come segue:

```
void Gruppo(void (*f)(), double x)
{static int a=0; f(&a,x);}
```

eseguendo poi `Gruppo(fk,x);`.

Quando le tre funzioni hanno tipo di risultato e numero e tipo dei parametri diversi, il procedimento diventa più difficoltoso (si possono usare puntatori `void *` e funzioni con un numero variabile di argomenti e altri accorgimenti), ma vale sicuramente la pena perchè risparmia tutto il lavoro successivo che nascerebbe dalla presenza di variabili globali.

La seguente funzione calcola la lunghezza della stringa che corrisponde al numero reale `x` con il formato `Formato`:

```
int Lungstringareale(double x, char *Formato)
{static char a[100]
sprintf(a,Formato,x); return strlen(a)}
```

CAPITOLO 9

Funzioni con un numero variabile di argomenti

Una funzione con un numero variabile di argomenti deve avere la seguente forma:

```
tipo1 f(tipo2 a, ...)
{va_list lista; ...
va_start(lista,a); //con variazioni
...
x=va_arg(lista,tipo);
...
va_end(lista);}
```

I puntini nella lista degli argomenti (dopo `tipo2 a`) devono effettivamente essere scritti così; gli altri puntini all'interno del corpo della funzione, invece, indicano parti da completare; `tipo` è il tipo della variabile che viene prelevata; la funzione `va_arg` può essere chiamata più volte e non è necessario che il tipo prelevato sia sempre lo stesso. Ogni chiamata di `va_arg` preleva la prossima variabile dalla lista degli argomenti (da sinistra a destra cominciando con la variabile che segue la variabile con cui abbiamo inizializzato la lista mediante `va_start` (nel nostro caso `a`). Non dimenticare `va_end` alla fine altrimenti si avrà quasi certamente un errore. Queste istruzioni richiedono l'header `<stdarg.h>`.

Vediamo ora come funziona `va_start`.

Tra gli argomenti della funzione ci deve essere almeno una variabile di tipo noto; ce ne possono essere anche più di una e una di esse deve essere il secondo argomento di `va_start`; le variabili introdotte successivamente al posto dei puntini vengono collocate in memoria dopo le variabili di tipo noto. `va_start` può essere chiamata anche più volte.

```
double somma(int n,...)
{va_list lista; int k; double s;
va_start(lista,n);
for(s=0,k=0;k<n;k++) s+=va_arg(lista,double);
va_end(lista); return s;}
```

La funzione viene chiamata nel modo seguente:

```
printf("%.2f\n", somma (5, 3.0, 2.0, 1.0, 4.0, 8.0));
```

Qui, come sempre nel C, quando il compilatore si aspetta un valore di tipo `double`, un valore intero deve essere convertito; ad esempio, come abbiamo fatto qui, scrivendolo come numero decimale con almeno una cifra dopo il punto decimale.

Creiamo adesso una funzione che permette di congiungere un numero variabile di stringhe (sostituendo in parte `sprintf`) dopo esecuzione della

funzione la stringa concatenata si trova nell'indirizzo indicato dal primo argomento. L'elenco delle stringhe da concatenare deve terminare con `NULL`. Lo spazio necessario per raccogliere il risultato (cioè la stringa concatenata) deve essere riservato in precedenza.

Esempio di come la funzione viene usata (*):

```
char a[1000];
//A, B
//siano stati dichiarati in precedenza
//e gli siano stati assegnati valori
Concatena(a,A,"/",B,"-35.84",NULL);
//equivalente a sprintf(a,"%s/%s-35.84",A,B);
```

Dopo di ciò, se `A` contiene la stringa `"ufficio"` e `B` la stringa `"lettere"`, la stringa in `a` sarà: `"ufficio/lettera-35.84"`.

Definiamo la funzione `Concatena`:

```
void Concatena(char *E,char *A,...)
{va_list argo; int m,p; char *X;
va_start(argo,A);
for(p=0,X=A;X;) {m=strlen(X);memmove(E+p,X,m);
p+=m; X=va_arg(argo,char*);}
E[p]=0; va_end(argo);}
```

In (*) abbiamo riservato 1000 caratteri per `a`. Come facciamo però nel caso generale a sapere quanti caratteri servono? A questo scopo definiamo una funzione che conta i caratteri complessivi in una stringa molto simile a `Concatena`:

```
int Contacaratteri(char *A,...)
{va_list argo; int p; char *X;
va_start(argo,A);
for (p=0,X=A;X;){p+=strlen(X); X=va_arg(argo,char*);}
va_end(argo);return p;}
```

Adesso invece di (*) potremmo scrivere

```
char *E;
E=malloc(contacaratteri(A,"/",B,"-35.84",NULL)+1;)
Contacaratteri(E,A,"/",B,"-35.84",NULL);
```

Per non dover riscrivere tutti gli argomenti possiamo definire una funzione che conta i caratteri complessivi delle stringhe che poi concatena. A questo scopo usiamo la possibilità di ripetere `va_start`. La funzione che vogliamo definire presenta un'altra peculiarità: siccome la stringa che corrisponde alla concatenazione viene creata all'interno della funzione e deve essere esportata, dobbiamo usare un puntatore ad essa. Il primo argomento di `Concatenaconspazio` è quindi del tipo `char **` che abbiamo visto nel paragrafo dedicato al passaggio dei parametri.

```
void Concatenaconspazio(char **HE, char *A,...)
{va_list argo; int m,p; char *X;
va_start(argo,A);
for(p=0,X=A;X;){p+=strlen(X); X=va_arg(argo,char *);}
char *E=malloc(p+1);
va_start(argo,A);
```

```
for(p=0,X=A;X;){m=strlen(X); memmove(E+p,X,m);
p+=m; X=va_arg(argo,char *);}
E[p]=0; *HE=E; va_end(argo);}
```

La funzione deve essere usata così:

```
char *A="ufficio", *B="lettera";
char *E;
Concatenaconspazio(&E,A,"/",B,"35.84",NULL);
puts(E);
```

Se vogliamo un output diverso basta modificare:

```
char *Concatenaconspazio(char *A,...)
{...
E[p]=0; va_end(argo); return E;}
```

CAPITOLO 10

Strutture

Con `typedef` si possono assegnare nomi nuovi a tipi già esistenti, ad esempio `typedef size_t tipoindirizzo;`. Dopo di ciò invece di `size_t n;` possiamo scrivere `tipoindirizzo n;`. `typedef` viene spesso usato per abbreviare nomi di tipi complicati come faremo adesso per le strutture e più avanti con `typedef char *Char;` per i puntatori a caratteri.

Il modo più conveniente per definire strutture composte nel C è questo:

```
typedef struct {double x,y,z;} vettore;
```

Adesso, dopo la dichiarazione `vettore v;` i componenti di `v` sono `v.x`, `v.y` e `v.z`. Possiamo così definire una funzione per l'addizione di due vettori il cui risultato è ancora un vettore:

```
vettore somma(vettore v, vettore w)
{vettore s;
s.x=v.x+w.x; s.y=v.y+w.y; s.z=v.z+w.z;
return s;}
```

Se `A` è un puntatore e ha una struttura per la componente `x` di `*A`, cioè per `(*A).x`, si può anche scrivere `A->x` (Componente `x` del puntatore `A`; se `A` è double `A->x` è double). Non sarebbe corretto `*A.x` che viene interpretato come `*(A.x)`.

Nella nota 1.7 del corso di programmazione abbiamo introdotto il metodo del sistema di primo ordine per i numeri di Fibonacci:

$$\begin{cases} x_{n+1} = x_n + y_n & \text{con } n \geq 0 \\ y_{n+1} = x_n & \text{con } n \geq 0 \\ x_0 = 1 \\ y_0 = 0 \end{cases}$$

Definiamo una struttura per coppie di numeri con:

```
typedef struct {double x,y;} coppia;
```

e una funzione che corrisponde al sistema:

```
coppia Fibonacci(int n)
{coppia u,v;
if (n==0) {v.x=1; v.y=0;} else
{u=Fibonacci(n-1); v.x=u.x+u.y; v.y=u.x;}
return v;}
```

Per provare usiamo:

```
for (n=90;n<=100;n++)
printf("%3d %-12.0f\n", n, Fibonacci(n).x);
```

La risposta è fulminea! Le strutture sono molto comode, ma si rischia di imboccare una strada di eccessiva strutturazione e quindi scivolare verso una programmazione orientata agli oggetti. La filosofia alternativa è di lavorare con funzioni ben organizzate e limitare il lato personalizzabile alla buona organizzazione di funzioni.

CAPITOLO 11

Vettori di parole

Introduciamo le seguenti abbreviazioni:

```
typedef char *Char;
// vdp = vettore di puntatori;
// Vdp = puntatore a vdp;
typedef struct {Char *L; int m, sep, comm, cont, uni;} vdp;
// *L = lista di puntatori;
// m = lunghezza della lista;
// uni = righe che vanno lette insieme;
// comm = commento
// cont = continuazione
typedef vdp *Vdp;
```

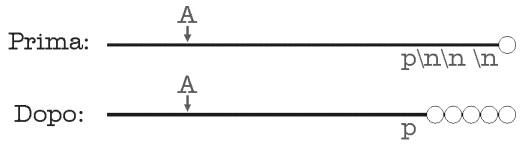
1. Alcune funzioni ausiliarie

`App(A,c)` numero delle comparizioni del carattere `c` nella stringa `A` (`c` viene rappresentato da un intero).

```
int App(Char A, int c)
{int n; for(n=0; *A; A++) if(*A==c) n++; return n;}
```

Vogliamo eliminare tutti i caratteri vuoti alla fine di una stringa `A` (tranne lo `0` terminale), cioè tutti i caratteri c con $-1 \leq c \leq 32$.

`Rid(A);` effettua questa riduzione.



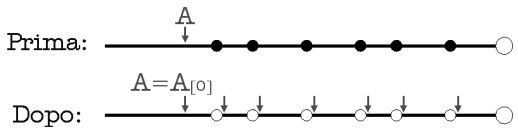

```
void Rid(Char A)
{if(!*A) return;
for (A=strchr(A,0)-1;-1<=*A&&*A<=32;A--) *A=0;}
```

Per creare un `Vdp` vuoto (cioè senza contenuto definito) di `m` elementi usiamo `P=Tvv(m);`[1]. Dove la funzione `Tvv` è così definita:

[1]Tvv sta per Testo Vettore Vuoto. In generale ci sono molte tecniche per ordinare biblioteche abbastanza grandi di funzioni: noi adottiamo quella di usare la prima lettera per definire l'ambiente in cui stiamo lavorando (Testo - Stringhe) e le successive per descrivere in modo sintetico cosa fa la funzione. Se questo ordinamento viene applicato sistematicamente permette di creare buone e vaste librerie di funzioni che possono evitare l'uso della programmazione orientata agli oggetti.

```
Vdp Tvv(int m)
{if (!m) return NULL;
Vdp P=malloc(sizeof(vdp));
p->L=malloc(m*sizeof(Char));
P->m=m; return P;}
```

Adesso sia data una stringa suddivisa in segmenti da un carattere separatore `sep` indicato con •:



con `P=Tvp(A);` creiamo un puntatore a questi segmenti. All'inizio la stringa viene ridotta con `Rid` poi mentre vengono creati i puntatori i caratteri separatori vengono sostituiti da `0` e le stringhe così determinate a loro volta ridotte. In questo modo naturalmente la stringa `A` viene modificata.

```
void Tvp(Char A, int sep)
{Char X,Y; int k; Vdp P;
Rid(A); if(!*A) return NULL; P=Tvv(App(A,sep)+1);
for (X=A, k=0;;X++, k++)
{P->L[k]=X; Y=X; X=strchr(X,sep); if(!X) return P;
*X=0; Rid(Y);}}
```

Si noti che `Tvp("",sep)` è il puntatore nullo, così come `Tvp("  ",sep);`.

2. Funzioni di output

`Tvo(P,Formato);` esegue l'output formattato per gli elementi di un `Vdp P;`. Ogni elemento `X` viene stampato come da `printf(Formato, X)`.

```
void Tvo(Vdp P, Char Formato)
{int i; Char A; if(!P) return;
for(i=0;i<P->m;i++)
{A=P->L[i]; if (A) printf(Formato,A);
else printf(Formato,"Nullo");}}
```

E' quindi previsto che una componente di `P` sia il puntatore nullo anche se ciò non accade se `P` è stato creato con `Tvp`. Esempio:

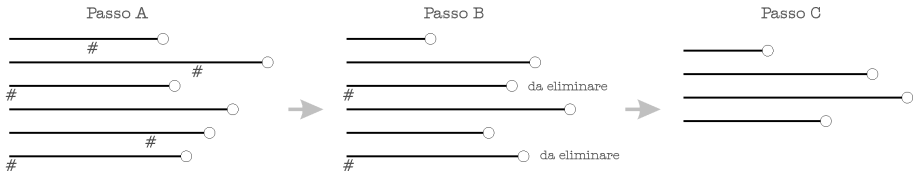
```
void Prova()
{Char A="alfa\nbeta\ngamma\n   \ndelta\n";
Vdp P=Tvp(A,'\n'); Tvo(P,"[%s]\n");}
// output: [alfa]
           [beta]
           [gamma]
           []
           [delta]
```

3. I commenti

Consideriamo adesso una stringa A che può contenere commenti definiti da un carattere `com` nello stesso modo in cui `#` definisce i commenti in Perl. Per eliminare inoltre del tutto le righe che iniziano con `com` procediamo così:

1. se `com == 0` allora non succede nulla;
2. il testo `A` viene trasformato in un `Vdp P` quindi i separatori non sono influenzati dai commenti;
3. tutti i `com` che non appaiono all'inizio di un elemento di `P` vengono sostituiti dal carattere `0`;
4. le componenti che iniziano con `com` vengono eliminate da `P`;
5. usando `realloc` viene eventualmente diminuita la memoria a disposizione di `P`; per questa ragione il primo argomento della seguente funzione `Tvsc` è del tipo `Vdp *`.

Questa procedura può essere così schematizzata:



La funzione che opererà nel modo appena visto sarà quindi:

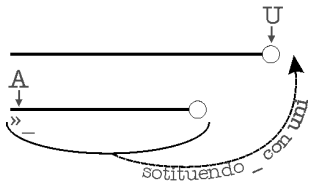
```
void Tvsc(Vdp *HP, int com)
{if(!com) return; Vdp P=*HP; Char X,A; int i,k;
for(i=0;i<p->m;i++){A=P->L[i]; X=strlen(A,com); if(X>A) *X=0; Rid(A);}
for(i=k=0;i<P->m;i++) if(*P->L[i]!=com) P->L[k++]=P->L[i];
P->m=k; if(!k) *HP=NULL; else P->L=realloc(P->L,k*sizeof(Char));}
```

Esempio:

```
void Prova()
{Char A="#commento+alfa+beta#commento+gamma+   +#commento+delta+eps#ilon\n";
Vdp P=Tvp(A,'+'); Tvsc(&P,'#'); Tvo(P,"[%s]\n");}
// output: [alfa]
           [beta]
           [gamma]
           []
           [delta]
           [eps]
```

4. Trattamento di continuazioni

`P` sia un `Vdp` da cui abbiamo eliminato i commenti con `Tvsc`. Il procedimento è diverso da quello visto in precedenza; stavolta `cont` sia un carattere (ad esempio `>` o meglio `»` = ALT + 187) che quando si trova all'inizio di un segmento di `P` ed è seguito da `_` (SPAZIO) indica che questo elemento deve essere unito a quello precedente mediante un carattere di unione `uni`.



`U` punta al punto di unione; » sparisce; l'unione avviene attraverso `strcat`. Anche in questo caso lo spazio superfluo generato dall'eliminazione di elementi di `P` viene liberato effettuando `realloc`. Se `cont = 0` non succede nulla.

```
void Tvu(Vdp *HP, int cont, int uni)
{if(!cont) return; Vdp P=*HP; int i,k,m; Char A,U;
U=strlen(P->L[0],0); for(i=1,k=0;i<P->m;i++)
{A=P->L[i]; if(*A!=cont) P->L[++k]=A;
else{strcat(p->L[k],A+1); *U=uni;} U=strchr(A,0);}
m=P->m=k+1; p->L=realloc(P->L,m*sizeof(Char));}
```

Esempio:

```
void Prova()
{Char A="Antonio+» Rossi+Maria Teresa+» Verdi,+» notaioio+Rosa Luppo";
Vdp P=Tvp(A,'+'); Tvu(&P,'»',' '); Tvo(P,"[%s]\n");}
// output: [Antonio Rossi]
           [Maria Teresa Verdi, notaio]
           [Rosa Luppo]
```

5. La funzione finale

Uniamo adesso queste funzioni in un'unica funzione `Tv` che crea un puntatore `P` da un testo `A` mediante `Tvp` eseguendo poi su `P` `Tvsc` per eliminare i commenti e `Tvu` per riunire gli elementi indicati.

```
Vdp Tv(Char A, int sep, int comm, int cont, int uni)
{Vdp P=Tvp(A,sep); if(!P) return NULL;
Tvsc(&P,comm); Tvu(&P,cont,uni);
P->sep=sep; P->comm=comm;p->cont=cont;
P->uni=uni; return P;}
```

Per liberare lo spazio occupato da un `Vdp` usiamo:

```
void Tvl(Vdp *HP)
{Vdp P=*HP; free (P->L; free(P);}
```

Esempio:

```
void Prova()
{Char A="Antonio+» Rossi+Maria Teresa+» Verdi,+» notaio+Rosa Luppo";
Vdp=Tv(A,'+','#','»',' '); Tvo(P,"[%s]\n"); Tvl((&P); puts("---");
A="alfa#commento+» e beta+#commento+gamma e+» delta+epsilon";
P=Tv(A,'+','#','»',' '); Tvv(P,"[%s]\n");}

//output [Antonio Rossi]
         [Maria Teresa Verdi, notaio]
         [Rosa Luppo]
         ---
         [alfa e beta]
         [gamma e delta]
         [epsilon]
```

Il C permette di definire costanti (e in verità anche macroistruzioni dipendenti da parametri) mediante `# define`, un'istruzione per il preprocessore. Ad esempio:

```
# define pi 3.14159
# define cont '»'
```

non richiede nessun segno di uguaglianza.

CAPITOLO 12

Lettura e scrittura di file

E' ovviamente importante possedere funzioni per la lettura e la scrittura di file e per il catalogo di una cartella. Per la piena comprensione delle funzioni qui indicate bisogna consultare i manuali per la programmazione di sistema soprattutto *W. R. Stevens* 'Advanced programming in the Unix enviroments' Addison-Wesley 1992.

Le nostre funzioni richiedono gli header

```
<sys/stat.h>
<sys/types.h>
<fcntl.h>
<unistd.h>
```

1. Per calcolare la lunghezza di un file:

```
size_t lunghezzafile(Char A)
{struct stat att; int e;
e=stat(A,&att); if(e<0) return -1;
if(att.st_mode && S_IFREG) return att.st_size; return -1;}
```

2. Per leggere un file usiamo:

```
void leggifile(Char A, Char Testo)
{int fk; size_t n;
fk=open(A,O_RDONLY); if(fk<0) {*Testo=0; return 0;}
n=lunghezzafile(A); read(fk,Testo,n); close(fk); Testo[n]=0;}
```

3. Esempio:

```
void Prova()
{Char Nome="lettera";
int n=lunghezzafile(Nome); Char Testo=malloc(n+4);
leggifile(Nome,Testo); puts(Testo); free(Testo);}
```

4. Per la scrittura su un file:

```
void scriviinfile(Char A,Char Testo)
{int fk;
fk=open(A,O_WRONLY|O_CREAT|O_TRUNC|O_SYNC,S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
write(fk,Testo,strlen(Testo));close(fk);}
```

5. Per aggiungere un testo a un file (senza cancellare il testo presente):

```
void aggiungiafile(Char A, Char Testo)
{int fk; fk=open(A,O_WRONLY|O_APPEND|O_CREAT);
write(fk,Testo,strlen(Testo));close(fk);}
```

Esecuzione di una funzione f per ogni elemento di una cartella. Richiede l'header `<dirent.h>`.

```
void esegui(void f(),Char Cartella)
{DIR *Dir; struct dirent *Info;
Dir=opendir(Cartella); if(!Dir) return;
while(Info=readdir(Dir)) f(Info->d_name); closedir(Dir);}
```

Una stranezza: in alcuni compilatori `I` non può essere usata come nome di variabile o di funzioni.