Matrici, stringhe e I/O da file

- Memory leakage
- Matrici
- Stringhe
- I/O da file

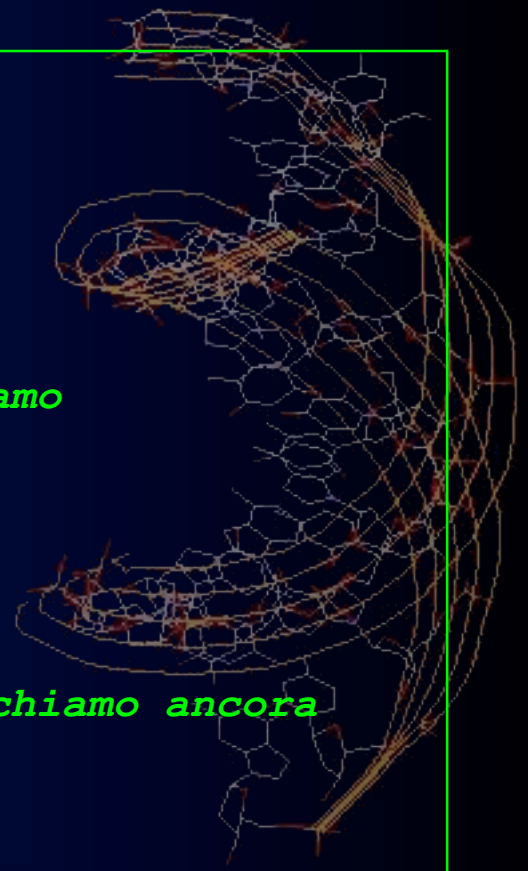
Memory Leakage (I)

Se si “perde” il valore di un puntatore di una zona di memoria allocata, quella memoria non e’ piu’ utilizzabile dal processo e non e’ piu’ “liberabile”: rimane quindi proprieta’ del processo fino alla sua terminazione.


```
main()
{
  const int dim=10;
  int *b;
  b =(int *)MyMalloc(dim*sizeof(int)); //allochiamo
  printf("\n Indirizzo di b   : %x", &b);
  printf("\n Indirizzo di b[0]: %x", &b[0]);
  b = (int *)MyMalloc(2*dim*sizeof(int)); //allochiamo ancora
  printf("\n Indirizzo di b   : %x", &b);
  printf("\n Indirizzo di b[0]: %x", &b[0]);
  free(b);//libera la memoria allocata
}
```

Memory Leakage (II)

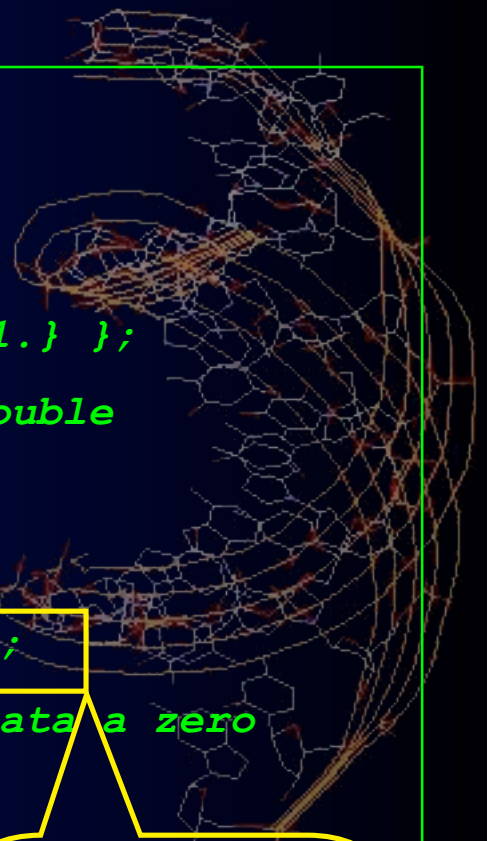
```
main()
{
  const int dim=10;
  int *b;
  b =(int *)MyMalloc(dim*sizeof(int)); //allochiamo
  printf("\n Indirizzo di b   : %x", &b);
  printf("\n Indirizzo di b[0]: %x", &b[0]);
  free(b);
  b = (int *)MyMalloc(2*dim*sizeof(int)); //allochiamo ancora
  printf("\n Indirizzo di b   : %x", &b);
  printf("\n Indirizzo di b[0]: %x", &b[0]);
  free(b);//libera la memoria allocata
}
```



Array di Puntatori

Una matrice bidimensionale può essere rappresentata come un vettore a due indici, o come un vettore di puntatori

```
main(){
  const int dim = 3 ;
  int i,j;
  double a[3][3]= { {1.,0.,0.},{0.,1.,0.},{0.,0.,1.} };
  double *b[dim];/* a è una matrice quadrata di double
        b è un array di puntatori */
  for(i=0;i<dim;i++) {
        b[i]=(double *)calloc(dim ,sizeof(double));
        }//ora b è una matrice quadrata inizializzata a zero
  for(i=0;i<dim;i++){
        for(j=0;j<dim;j++) {
        b[i][j]=a[i][j] ;   }
 }
}
```



Un vettore di double per ciascun puntatore

Puntatori a Puntatori

Un array di puntatori è…un puntatore a puntatori !

```
main(){
  const int dim = 3 ;
  int i,j;
  double a[3][3]= { {1.,0.,0.},{0.,1.,0.},{0.,0.,1.} };
  double **b;  // b è un puntatore ad un puntatore ad un double
  b=(double **)calloc(dim,sizeof(double *))
  for(i=0;i<dim;i++) {
        b[i]=(double *)calloc(dim ,sizeof(double));
        }//ora b è una matrice quadrata inizializzata a zero
  for(i=0;i<dim;i++){
        for(j=0;j<dim;j++) {
        b[i][j]=a[i][j] ;   }
 }
}
```

Vettore di puntatori, uno per riga

Matrici come argomento di funzioni

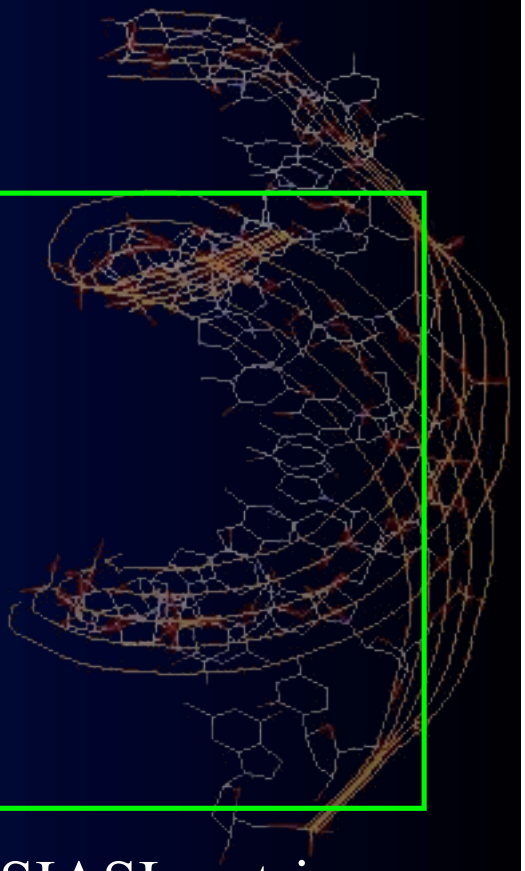
I puntatori a puntatori sono spesso l'unico mezzo pratico di passare matrici come argomenti di funzioni. Ciò accade perché il compilatore ha bisogno che nel prototipo e nella definizione della funzione venga specificata ogni dimensione di un array successiva alla prima. Ad esempio volendo costruire una funzione per calcolare la trasposta si avrebbe:

```
void  trasponi3{double a[][3],double at[][3]) {
  const int dim = 3 ;
  int i,j;
  for(i=0;i<dim;i++){
        for(j=0;j<dim;j++) {
           at[i][j]=a[j][i]}
  }
}
```

..e se volessi calcolare la trasposta di una matrice 4X4 ??

Esempio

```
void  trasponi4{double a[][4],double at[][4]) {
  const int dim = 4 ;
  int i,j;
  for(i=0;i<dim;i++){
        for(j=0;j<dim;j++) {
           at[i][j]=a[j][i]}
  }
}
```



…oppure, per poter calcolare la trasposta di QUALSIASI matrice…

Esempio II

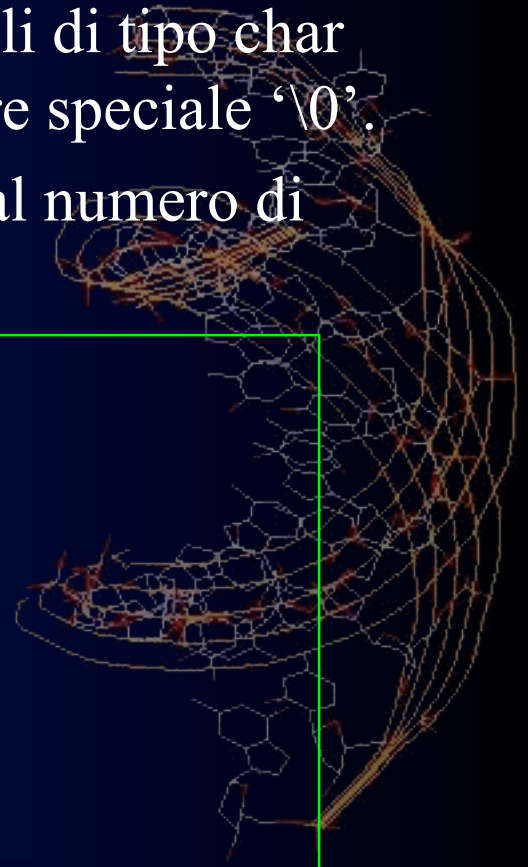
```
void  trasponi (double **a,double **at,int dim)
{
  /* a e at sono matrici >*dinamiche*< allocate altrove */
  int i,j;
  for(i=0;i<dim;i++){
        for(j=0;j<dim;j++) {
           at[i][j]=a[j][i]
        }
  }
}
```

Stringhe

In C una stringa è semplicemente un array di variabili di tipo char terminata nella rappresentazione interna dal carattere speciale '\0'.

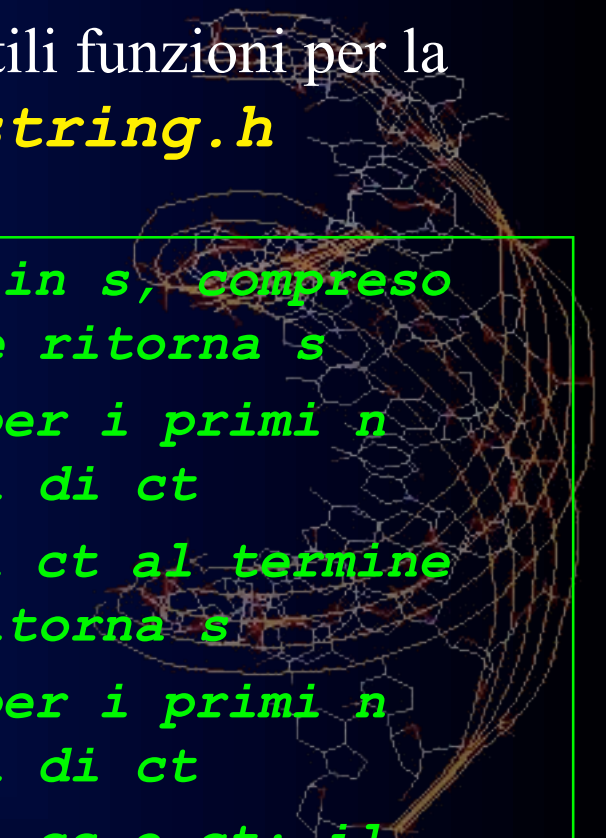
L'array contiene quindi un elemento in più rispetto al numero di caratteri.

```
#include<stdio.h>
  main(){
  char *s1, s2[]="Hello world!", *s3;
  s1=s2;
  s3="Hello again!";
  printf("%s\n%s\n%s\n",s1,s2,s3);
  *(s1+1)='a';
  printf("%s\n%s\n%s\n",s1,s2,s3);
}
```



Manipolazione di stringhe

La libreria standard del C fornisce una serie di utili funzioni per la manipolazione di stringhe, definite nell'header ***string.h***



`char *strcopy(s, ct)`	copia ct in s, compreso lo '\0' e ritorna s
`char *strncopy(s, ct, n)`	idem ma per i primi n caratteri di ct
`char *strcat(s, ct)`	concatena ct al termine di s e ritorna s
`char *strncat(s, ct, n)`	idem ma per i primi n caratteri di ct
`int strcmp(cs, ct)`	confronta cs e ct; il risultato è <0 se cs<ct, 0 se cs==ct, >0 se cs>ct
`size_t strlen(cs)`	ritorna la lunghezza di cs

Esempio

```
#include<stdio.h>
#include<string.h>
  main(){
  char *s1, s2[]="Hello", *s3;
  s1 = " world !";
  s3 = " again!";
  s3 = strcat(s2,s3);
  s1 = strcat(s2,s1);
  printf("%s\n%s\n",s1,s3);
  if(strcmp(s1,s3)>0)
        printf("%s precede %s\n",s1,s3);
  else
         printf("%s precede %s\n",s3,s1);
}
```

Gestione dei files

Il C permette di gestire i files sia ad “alto livello”, tramite puntatori a strutture di tipo FILE che “a basso livello”, più direttamente collegato al sistema operativo, tramite i cosiddetti descrittori di file. Ci concentreremo sull’accesso “ad alto livello”.

Per poter accedere ad un file da un programma C la prima operazione da effettuare è l’apertura del file. La funzione di libreria fopen, definita in stdio.h accetta in ingresso un nome file, esegue le necessarie trattative col sistema operativo e restituisce un puntatore a una struttura di tipo FILE che deve essere utilizzata nelle successive operazioni sul file stesso. In questo modo si ha accesso allo stream di bytes rappresentato dal file stesso.

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

Apertura dei files

Se il file che stro cercando di aprire in lettura non esiste (ad es. ho sbagliato a scrivere il nome) o non ho il permesso di aprire in scrittura il file di output la chiamata ad fopen restituisce un puntatore a NULL. In tal caso dovrei segnalare un errore ed uscire. Posso farlo ad es. reidfinendo una opportuna funzione MyFopen in analogia con quanto fatto per MyMalloc:

```
FILE *MyFopen(char *name, char *mode) {

FILE *f;
f = fopen(name,mode);
if(f==NULL) {
   fprintf(stderr,"Error opening file %s",name);
   exit(1);
}
return f;
}
```

Esempio

```
#include <stdio.h>

int main(){
 FILE *infile, *outfile, *logfile, *tempfile;

 infile   = MyFopen("dati.dat","r");
 /* infile è il puntatore ad un file di nome dati.dat
aperto in sola lettura ("r")*/
 outfile  = MyFopen("risultati.txt","w");
/* w sta per write (scrittura) */
 logfile  = MyFopen("registro.log","a");
/* a sta per append (aggiunta in coda)*/
 tempfile = MyFopen("tmpfile.dat", "w+");
/* w+ (o r+) sta per lettura-scrittura */
…
return 0;
}
```

fprintf e fscanf

Le funzioni principali per l'I/O formattato da file sono :

```
int fprintf(FILE * fp, const char  *format,…);
int fscanf(FILE * fp, const char  *format,…);
```

Il primo argomento è un puntatore a FILE (ad es. il risultato di una chiamata a fopen). Gli altri argomenti (la stringa di formato e la sequenza di argomenti opzionali) seguono le stesse identiche convenzioni usate per printf e scanf. In effetti il C mette a disposizione tre puntatori *stdin*, *stdout*, *stderr* per i file standard di I/O. Si ha allora :

```
printf("Hello");     ⇔   fprintf(stdout,"Hello");
scanf("%d",&i);      ⇔   fscanf(stdin,"%d",&i);
```

Altre funzioni di I/O su file

Funzioni per l'I/O di singoli caratteri:

```
int fgetc(FILE * fp);
int fputc(int c, FILE * fp);
```

Funzioni per l'I/O diretto di bytes senza conversioni:

```
size_t fread(void *pt,size_t dimensioneelemento, size_t
        numeroelementi, FILE * fp);

size_t fwrite(const void *pt,size_t dimensioneelemento,
        size_t numeroelementi, FILE * fp);
```

End of File e condizioni di errore

Nelle operazioni di accesso a file possono verificarsi delle condizioni di errore: ad esempio una chiamata a fread può restituire un numero di elementi letti inferiore a quello previsto se viene incontrata la fine del file (EOF) prima di aver letto tutti gli elementi.

Parimenti, fwrite può scrivere un numero inferiore di elementi se si verifica una condizione di errore nell'accesso al file. Per controllare il verificarsi di condizioni di end of file o di errore si usano le funzioni:

```
int feof(FILE * fp);
int ferror(FILE * fp);
```

Esse ritornano un valore *diverso da zero* se trovano alzata la flag di fine file e quella di errore rispettivamente per lo stream fp.

Posizionamento su file

```
long ftell(FILE * fp);
int fseek(FILE * fp, long offset, int place);
void rewind(FILE *fp);
```

ftell restituisce la posizione corrente all'interno del file.

fseek posiziona l'indice di posizione del file a *offset* bytes da *place* , che può assumere i valori `SEEK_SET` (inizio file), `SEEK_CUR` (posizione corrente) oppure `SEEK_END`(fine file). Se il file è un file di testo fseek va usata solo con place=`SEEK_SET` e offset ottenuto da una precedente chiamata a ftell.

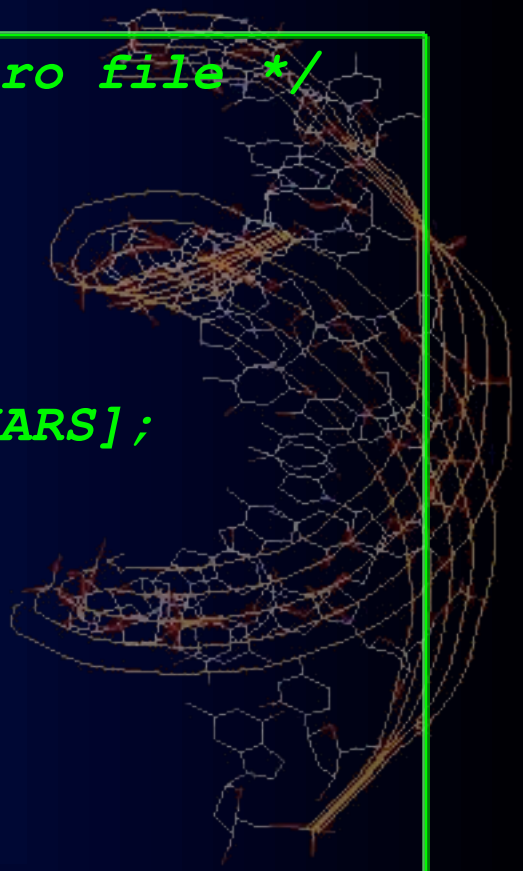
rewind posiziona l'indice del file a inizio file:

`rewind(fp)` ⇒ `fseek(fp, 0L, SEEK_SET)`

Un esempio

```
/* scrive un file al contrario in un altro file */
#include <stdio.h>
int main(){
 FILE *ifp, *ofp;
 const int MAXCHARS=100;
 char inp_name[MAXCHARS], out_name[MAXCHARS];
 int c;

 printf("\nNome file in ingresso\n");
 scanf("%s",inp_name);
 printf("\nNome file in uscita\n");
 scanf("%s",out_name);
 ifp    = MyFopen(inp_name,"r");
 ofp    = MyFopen(out_name,"w");
/* segue…*/
```



...segue

```
/* …segue */
fseek(ifp,0,SEEK_END); /* si sposta alla
                          fine del file */
fseek(ifp,-1,SEEK_CUR); /* si sposta indietro di un
                           byte = 1 carattere */
while(ftell(ifp)>0){
 c=fgetc(ifp) ; /*legge il carattere e sposta in avanti
                  il puntatore di posizione file */
 fputc(ofp,c); /* stampa il carattere sull'output */
 fseek(ifp,-2,SEEK_CUR); /* torna indietro di due
                            caratteri*/
 }
 fclose(ifp);  /* chiude il file in ingresso */
 fclose(ofp);  /*chiude il file di output */
 return 0;
}
```

Interfaccia con il sistema operativo

La gestione dei file è uno di quei casi in cui il linguaggio C si trova a comunicare più strettamente con il sistema operativo.

Una funzione molto generale e molto utile per comunicare con il sistema è la seguente :

```
int system(char *command);
```

Il controllo viene passato al sistema operativo, che interpreta la stringa *command* come un comando, lo esegue e ritorna il controllo al programma C.

Ad es. in UNIX system(“date”) causa la stampa della data e ora corrente immagazzinata nell’OS. L’uso di system va fatto con molta attenzione nei casi in cui sia fondamentale la portabilità da un sistema all’altro.