Laboratorio di Metodi Numerici per Equazioni alle Derivate Parziali I a.a.2018/2019

Francesca Fierro

Email: francesca.fierro@unimi.it

Ricevimento:
Mercoledì 10.30 - 12.30
(o su appuntamento via email)

Programmare in C

Fase 1: Scrittura

- Un programma in linguaggio C, è costituito da funzioni e variabili contenute in uno o più files con estensione `.c`

- Una funzione contiene istruzioni che specificano operazioni da effettuare sui dati memorizzati nelle variabili.

- Ogni programma deve contenere la funzione main() che sarà la prima ad essere eseguita e chiamerà altre funzioni che possono appartenere al programma o alle librerie di sistema.

Tra le librerie standard ricordiamo:

```
stdio.h :   funzioni per I/O
stdlib.h :  funzioni per l'allocazione di
            spazi di memoria, etc.
math.h :    funzioni matematiche
```

per utilizzare una funzione di libreria questa deve essere inclusa nel codice con la direttiva `#include` del preprocessore, ad esempio:

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

Fase 2: Traduzione

Questa fase si articola a sua volta in tre passi
- **preprocessing** vengono eseguite direttive specifiche del preprocessore al fine di includere file, definire costanti simboliche e macro o gestire la compilazione condizionale del codice
- **compilazione** il codice viene tradotto in linguaggio macchina
- **linking** vengono risolti i riferimenti a funzioni e variabli definite ad esempio nelle librerie standard o altre definite dall'utente e viene prodotto un file eseguibile.

Fase 3: Esecuzione

Il più semplice programma C:

Fase 1: Scrittura

nel file `hello.c`

```
#include<stdio.h>
main()
{
printf(‘‘ciao mondo’’);
}
```

Fase 2: Traduzione

```
cc hello.c -o hello -lm
```

Fase 3: Esecuzione

```
./hello
```

Il preprocessore C

Il preprocessore C è un programma a se stante che viene eseguito dal compilatore prima della compilazione vera e propria, esso viene utilizzato principalmente ai seguenti scopi:

1. includere files

2. espandere in linea costanti simboliche e macro

3. valutare e gestire compilazioni condizionali

Il preprocessore legge un file sorgente C e produce in output un altro sorgente C dopo avere eseguito le operazioni sopra citate,

Le direttive per il preprocessore possono comparire in qualsiasi punto del codice ed iniziano con il carattere #.

1. `#include ‘‘nome-file’’`

   `#include <nome-file>`

   Nel primo caso il file da includere viene ricercato nella directory corrente e poi, se non trovato, nelle directory standard; nel secondo caso la ricerca inizia in alcune directory standard.
   Di solito si usa per includere un *header* file (ovvero un file di testo che contiene i prototipi delle funzioni definite nel relativo file .c).

2. `#define  nome testo da sostituire`

   permette di definire *macro* o *costanti simboliche* ovvero di associare ad un identificatore (nome) una stringa di testo che verrà sostituita dal precompilatore ad ogni occorrenza dell’identificatore presente nel sorgente.
   Le marco possono essere definite con o senza argomenti.

Esempi:

```
#define PI 3.14159
#define TRUE 1
#define SQR(a) ((a)*(a))
```

ATTENZIONE:

```
#define SQR(a) a*a
```

SQR(x+1) $\rightarrow$ x+1*x+1 $\neq$ (x+1)*(x+1)

3. Inclusione condizionale:

```
#if DIM == 2
   ...
#endif
```

se l'espressione `DIM==2` è diversa da 0 include le linee di codice che seguono

```
#if DIM == 2
   ...
#elif DIM == 3
   ...
#else
   ...
#endif
```

include diverse linee di codice a seconda dei casi DIM =2 ,3 o altro.

Variabili

In C ogni variabile è individuata da un *nome* ed è caratterizzata da un *tipo* e una *classe di memoria*.

Assegnare un **tipo** ad una variabile significa specificare l'insieme dei valori che essa può assumere.

La **classe di memoria** a cui appartiene una variabile ne determina il suo ciclo di vita e la sua visibilità (*scope*)

Scope = porzione di programma in cui un nome di variabile (o di funzione) può essere usato
Ciclo di vita = lasso temporale in cui una variabile (o una funzione) è accessibile.

Ogni variabile va dichiarata ed inizializzata

Tipi di dati fondamentali

`char`	`qualsiasi carattere (8 bits=1 byte)`
`int`	`numero intero (32 bits = 4 byte)`
`float`	`numero floating-point in singola precisone (32 bits = 4 byte)`
`double`	`numero floating-point in doppia precisone (64 bits = 8 byte)`
`void`	`un insieme vuoto di valori`

qualificatori di dimensione: `long, short`

qualificatori aritmetici: `signed, unsigned`

`short int`	`numero intero (16 bits = 2 byte)`
`long double`	`numero floating-point in quadrupla precisone (128 bits = 16 byte)`

`unsigned int`	`numero intero positivo`
`unsigned char`	`un carattere utilizzando valori interi tra 0 e 255`
`signed char(o char)`	`un carattere utilizzando valori tra -128 ..0 ..127`

Dichiarazioni di variabili

```
int i,j,n;
double x,y;
char c;
```

Inizializzazione

```
 int  x = 1;
```

oppure

```
 int x;
 x=1;
```

altri esempi

```
 char sl='\';
```

Classi di memoria delle variabili

- **variabili automatiche (o locali)**:
  - sono le variabili dichiarate dentro una funzione,
  - sono inizializzate con valori casuali.
  - *Ciclo di vita:* dal momento della chiamata della funzione fino al suo termine.
  - *Scope:* sono accessibili solo all'interno della funzione in cui sono definite

- **variabili esterne (o globali)**:
  - sono definite una sola volta al di fuori di qualsiasi funzione,
  - se non diversamente specificato sono inizializzate a zero.
  - *Ciclo di vita:* tutta la durata del programma.
  -*Scope:* sono accessibili in qualsiasi punto del file dove sono definite dopo la loro definizione, ed in qualsiasi altro punto del programma purché vengano dichiarate come `extern` nella funzione o nel file che le utilizza. Ad esempio:

```
extern int n;
```

Modificare il ciclo di vita: static

```
static int n;
```

- La dichirazione `static` di una variabile esterna o funzione limita lo scope al file sorgente in cui si trova

- Una variabile automatica dichiarata `static` non scompare al termine della chiamata della funzione in cui è definita.

Dichiarare costanti: const

Oltre che con la direttiva `#define` del preprocessore si possono definire identificatori per quantità costanti preponendo il qualificatore `const` davanti alla dichiarzione di una variabile per far si che il valore con cui viene inizializzata la variabile non venga più alterato.

Esempi:

```
const double   e = 2.71828182845905;
const char     msg[]= ‘‘warning’’;
```

Strutture dati: vettori o arrays

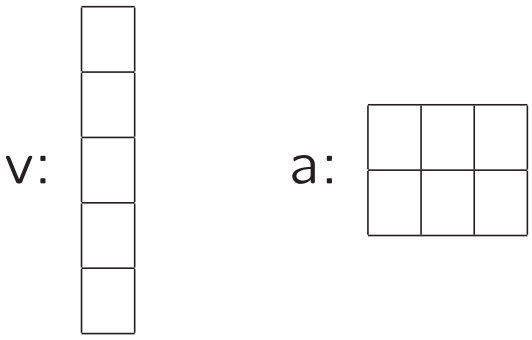
Una struttura dati si ottiene mediante composizione di altri dati. Permette di definire un tipo di dato derivato, ovvero costruito a partire da tipi di dato fondamentali.
Un esempio sono gli *arrays*, ovvero sequenze di lunghezza fissata di elementi dello stesso tipo.

La singola componente di un vettore è individuata da un indice di tipo intero che in C parte da 0.

In C non esistono operatori specifici per i vettori; per operare sui vettori è necessario operare singolarmente sulle signole componenti.

In C si possono definire array mono e bidimensionali (vettori e matrici) dichiarandoli (e inizializzandoli) ad esempio nel modo seguente

```
int       v[5];
double    mat[2][3];
```




```
double v[]={3.1, 5.2, 0, 1};
int mat[2][3]={{1, 2, 3},{4, 5, 6}};
char msg[]=''ciao'';
char msg[]={'c','i','a','o','\0'};
```

Funzioni

Una funzione consiste di un blocco di istruzioni che possono essere ripetute usando il nome della funzione ed i suoi parametri senza preoccuparsi dei dettagli implementativi.

Ogni funzione deve essere definita secondo il seguente *prototipo* :

```
tipo ritornato  nome-funzione( dichiarazioni
                               parametri     )
{
 dichiarazioni
 istruzioni
}
```

Il valore di ritorno viene resituito dalla funzione al chiamante tramite il comando
`return(espressione);`
oppure
`return;`
che compare nel corpo della funzione.

Esempio 1: la funzione minimale

```
void dummy(){}
```

Le definizioni di funzioni possono comparire in qualsiasi ordine all'interno di uno o più files ma:

- nessuna funzione può essere definita dentro un'altra funzione
- nessuna funzione può essere spezzata su più files
- prima di chiamare una funzione che non è stata ancora definita è bene dichiararla

Esempio di dichiarazione

```
double power(double x,int n);
double power(double ,int );
```

In C gli argomenti delle funzioni vengono passati *per valore*. Ovvero al momento della chiamata la funzione copia in una variabile locale il valore dell'argomento passato. Operando su tale copia non è possibile che una funzione modifichi una variabile della funzione chiamante.

Fa eccezione il caso di vettori e matrici.

È possibile realizzare in C anche il passaggio di parametri per "indirizzo", ovvero è possibile fare in modo che una funzione modifichi una variabile della funzione chiamante, passando ad una funzione come parametro in entrata l'indirizzo della variabile (tecnicamente il *"puntatore"*).

Puntatori

Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile

```
p= &c;
```

assegna l'indirizzo di `c` alla variabile `p`

```
*p
```

è il contenuto della cella di memoria puntata da `p`

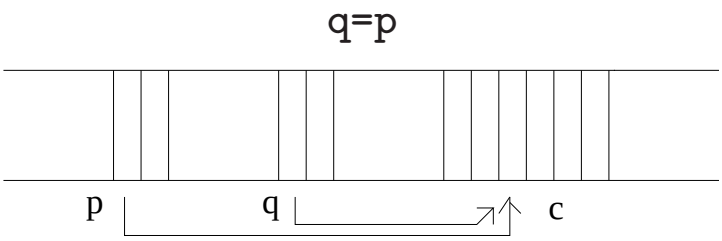
La dichiarazione di un puntatore include il tipo dell'oggetto a cui il puntatore è vincolato a puntare. Alcuni esempi sono:

```
int  *p;
double *q, func(char *);
```

```
int x=1, y=2;
int *p;

p=&x;       'p punta ad x'
y=*p        'y vale 1'
*p=0;       'x vale 0'

*p=*p+10;
y=*p+1;

*p+=1;   ++*p; (*p)++;

x=x+1;       <------>  x+=1;
se n=5;
x=n++;      ==>  x=5  n=6
x=++n;      ==>  x=6  n=6
```

Operazioni consentite sui puntatori

1. Assegnamento tra puntatori dello stesso tipo



2. Addizione e sottrazione tra puntatori ed interi

```
p+2;    p+n;
```

3. Confronto $==, !=, <, >, >=$ etc. fra puntatori ad elementi di uno stesso vettore

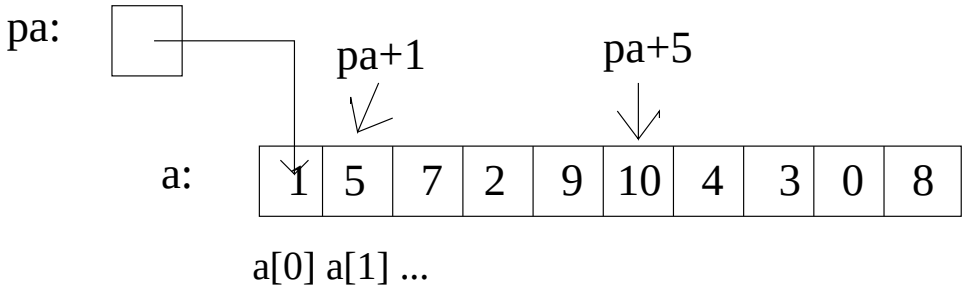
```
p<q;
```

4. Assegnamento e confronto con lo zero

```
p=0;    p=NULL;    p== 0;
```

Puntatori e vettori

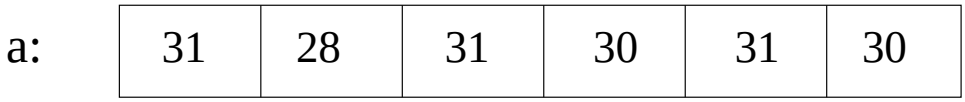
```
int a[10], *pa;
pa =&a[0];  (oppure pa=a;)
```



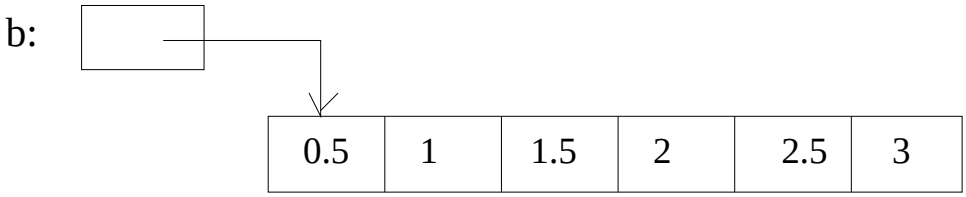

```
x= *pa;      'x vale 1'
pa+1;
*(pa+1);     vale 5

a[i]     <->    *(a+i)
&a[i]    <->    a+i
pa[i]    <->    *(pa+i)

pa=a; pa++;    sono legali
a=pa; a++;      sono illegali!
```

Memorizzare vettori

- `int a[6]={31,28,31,30,31,30}`



- `double *b;`
  `b=(double*)malloc(6*sizeof(double));`
  `b[0]=0.5; b[1]= ...; b[2]=...; ...`



Si differenziano principalmente nel modo in cui viene riservato spazio di memoria per contenere i valori dei coefficienti.

Memoria statica e dinamica

Il termine *allocazione* indica l'assegnazione di un blocco di memoria ad una variabile per memorizzarne il valore. Può avvenire i due modi: *statico* o *dinamico*.

L'**allocazione statica** della memoria permette di allocare memoria per una variabile le cui dimensioni sono note e fissate. La memoria è allocata sul segmento di memoria detto *stack* e non è modificabile.
L'**allocazione dinamica** della memoria permette di allocare memoria per una variabile le cui dimensioni possono essere modificate durante l'esecuzione del programma.
La memoria è allocata sul segmento di memoria detto *heap* facendone esplicitamente richiesta tramite apposite funzioni di libreria, e viene deallocata solo esplicitamente (non in automatico) per mezzo di altre apposite funzioni.
In C le principali funzioni atte a questo scopo sono `malloc` `realloc` e `free`.

`Esempio: free(b);`

Errori Comuni

Esempio 1: puntatore pendente ovvero un puntatore che si riferisce ad un'area di memoria non più valida, perchè già liberata

```
 float *ptr;
 ptr =(float *) malloc(10*sizeof(float));
// righe di codice in cui si usa ptr
free(ptr);
ptr[1]=4.5;
```

se si cerca di accedere a `ptr` senza riallocare nuova memoria può accadere che, se lo spazio di memoria a cui punta `ptr` è stato riallocato, l'assegnamento successivo avrà la conseguenza di " sporcare" i nuovi dati. Per questo è buona regola rissegnare a 0 il putatore quando la memoria viene deallocata.

```
free(ptr);
ptr=0;
```

Esempio 2: puntatore pendente

```
float x,
float *ptr=(float *) malloc(10*sizeof(float));
float *pptr=ptr;
// istruzioni
free(ptr);
x=pptr[0];
```

Esempio 3: memory leak o perdita di memoria dovuta alla mancata liberazione di memoria di variabili non più utilizzate.

```
float *ptr=(float *) malloc(100*sizeof(float));
ptr=0;
```

oppure

```
float *ptr= (float *) malloc(100*sizeof(float));
// istruzioni
*ptr= (float *) malloc(10*sizeof(float));
```

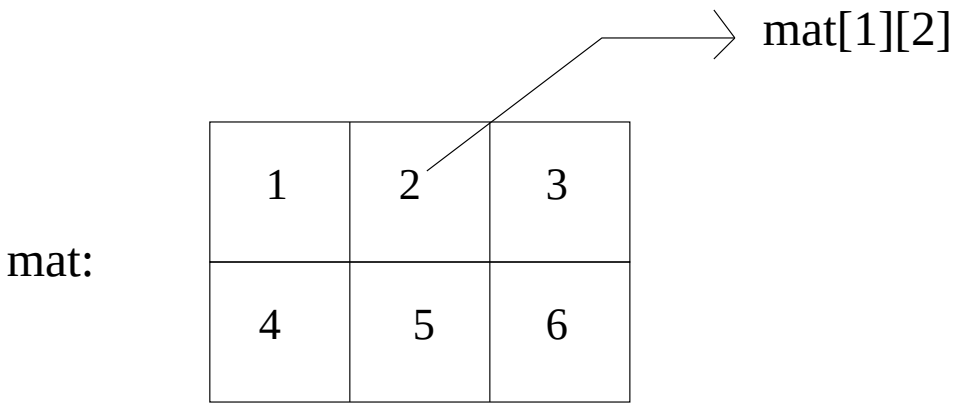
oppure

```
float *vec;
for(int i=0;i<1000;i++){
   vec=(float *) malloc(100*sizeof(float));
   //istruzioni che usano vec senza liberarlo
  }
```

Memorizzare matrici

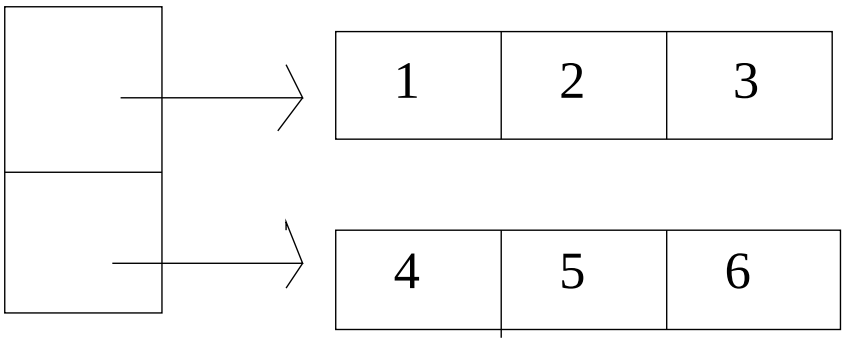
1. Vettori bidimensionali

```
int mat[2][3]={{1,2,3},{4,5,6}}
```



2. Vettori di puntatori

```
double * a[2];
a[0]=(double *)malloc(3*sizeof(double));
a[1]=(double *)malloc(3*sizeof(double));

a[0][0]=1; a[0][1]=2;  a[0][2]=3;
a[1][0]=4; a[1][1]=5;  a[1][2]=6;
```

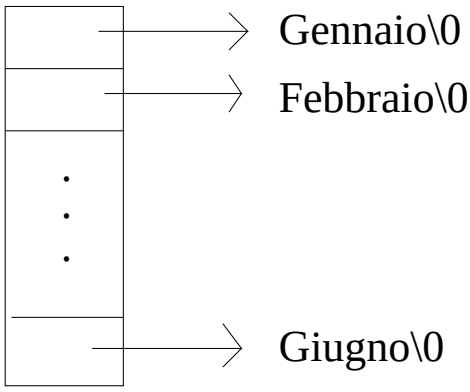


Per liberare la memoria allocata ad esempio

```
 free(a[0]);
```

```
a[i]:  puntatore a double
*a[i]: primo double del vettore puntato da a[i],
       ovvero a[i][0]
```

```
char * name[6]={ ‘‘Gennaio’’, ‘‘Febbario’’,
                 ‘‘Marzo’’, ‘‘Aprile’’,
                 ‘‘Maggio’’, ‘‘Giugno’’}
```

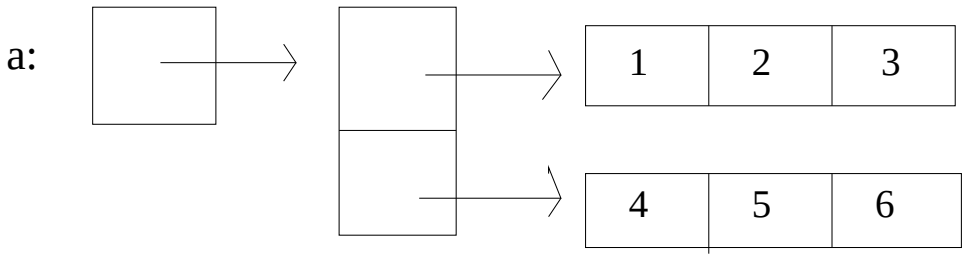


possibili righe di lunghezza diversa
(Matrici Sparse)

facili scambi di righe

3. Puntatori a puntatori

```
double ** a;

a=(double **) malloc(2*sizeof(double *));
a[0]=(double *)malloc(3*sizeof(double));
a[1]=(double *)malloc(3*sizeof(double));

a[0][0]=1; a[0][1]=2;  a[0][2]=3;
a[1][0]=4; a[1][1]=5;  a[1][2]=6;
```



Puntatori vettori ed argomenti di funzioni

Definizioni equivalenti di funzione che accettano in entrata un vettore sono:

```
int f(double v[]) {...}

int f(double *v) {...}
```

chiamate:

```
f( &a[2] )
f( a+2 )
```

Definizioni equivalenti di funzione che accettano in entrata una matrice sono:

```
int g(int mat[10][20]){...}

int g(int mat[][20]){...}

int g(int (*mat)[20]){...}
```

N.B. Nel caso di matrici è importante specificare sempre il numero di colonne.

Puntatori e funzioni

In C i puntatori vengono utilizzati per realizzare il passaggio di parametri *per indirizzo*.

Esempio: Se definisco la funzione

```
void swap (int x,int y)
{ int temp;
  temp=x;
  x=y;
  y=temp;
}
```

con la chiamata `swap(a,b);` non vengono scambiati i valori delle variabili a e b.
Viceversa se definisco:

```
void swap (int *px, int *py)
{ int temp:
  temp= *px;
  *px= *py;
  *py=temp;
}
```

effettuando la chiamata `swap(&a,&b);` i valori delle due variabili a e b vengono scambiati.

Anche il valore di ritorno restituito da una funzione può essere di tipo puntatore.
Quest'ultimo viene trasmesso per valore e quindi ne viene creata una copia nel programma chiamante, ciò garantisce che il puntatore sopravviva alla funzione anche se è stato creato all'interno del suo ambito.

Attenzione a non restituire puntatori a variabili locali.
Gli indirizzi locali sono indirizzi validi però individuano una zona di memoria che viene riusata. Di conseguenza può accadere che il valore puntato cambi in maniera inaspettata.

Esempio 1: Codice scorretto (ritorna un puntatore ad una variabile locale):

```
float *assign(int n){
 float b[20];
 for(int i=0;i<n;1++)
     b[i]=i;
 return b;
}
```

Codice corretto (ritorna un puntatore a memoria dinamica):

```
float *assign(int n){
 float *b=(float *) malloc(n*sizeof(float));
 for(int i=0;i<n;1++)
     b[i]=i;
 return b;
}
```

Puntatori a funzioni

Esempio di dichiarazione

```
int (*pfun) (double *,double *);
```

attenzione alle parentesi diverso è:

```
int *pfun(double *,double*);
```

Esempio di chiamata:

```
 (*pfun)(u,v);
```

dove

```
 double *u, *v;
```

Dichiarazioni complesse

```
int (*daytab)[13]
```

puntatore a vettore di 13 int

```
int *daytab[13]
```

vettore di 13 puntatori a int

```
void *comp()
```

funzione che ritorna un puntatore a void

```
void (*comp)()
```

puntatore a funzione che ritorna un void

```
char (*(*y())[])()
```

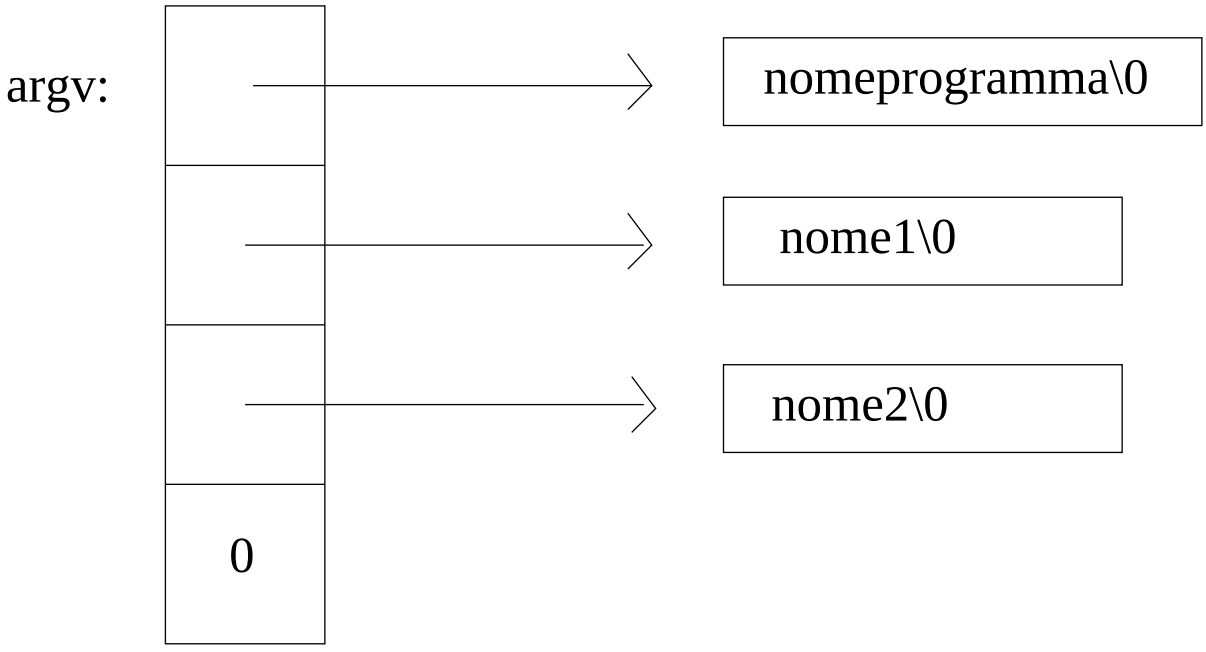
funzione che restituisce un puntatore ad un vettore di puntatori a funzioni che ritornano un char

```
char (*(*y[3])())[5]
```

vettore di 3 puntatori a funzioni che restituiscono un puntatore ad un vettore di 5 char

Argomenti per la funzione main

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



I/O

Stampa sullo standard output (video):

```
int printf(char *format, arg1, arg2,...)
```

ad esempio

```
printf("v[%d]=%g\n",i, v[i]);
```

Caratteri di conversione

`d` : intero

`c` : singolo carattere

`s` : stringa di caratteri

`f, e, g` : double

```
printf("x=%6.2f\n",x);
```

Lettura dallo standard input (tastiera):

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

ad esempio

```
int day,year;
char monthname[12];
scanf("%d %s %d\n",&day, monthname, &year);
```

I/O da/su file

Dichiarazione di file-pointer:

```
FILE *fp;
```

Per aprire un file utilizzare la funzione:

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

Esempio:

```
main (int argc, char * argv[])
{
   char name[80];

   if (argc <= 1)
     {
      printf ("data file name ");
      scanf ("%s", name);
     }
   else
      strcpy (name, argv[1]);
   printf("Nome file: %s\n", name);

   fp=fopen (name, "r");
  ......
}
```

Un file può essere aperto in lettura “r” in scrittura “w” in aggiunta o append “a”.

Per leggere dati da file:

```
int fscanf( FILE *fp, char * format, ...);
```

Per scrivere dati su file:

```
int fprintf( FILE *fp, char * format, ...);
```

Esempio:

```
 fprintf(fp,"%8e \t %e \n", x, y);
```

È buona regola chiudere un file quando non si usa più con la funzione

```
int fclose (FILE *fp);
```

ad esempio

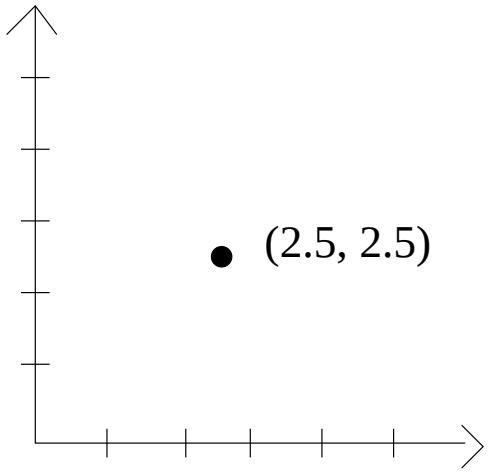
```
fclose (fp);
```

Strutture

Una struttura è un tipo di dato derivato che raggruppa elementi di uno o più tipi, sotto un unico nome comune

Definizione di una struttura

Esempio




```
struct point {
  double x;
  double y;
};
```

point è l'indentificatore (o tag) che attribuisce un nome alla struttura.
x,y sono i campi (o membri) della struttura.

Dichiarazione ed inizializzazione di variabili di tipo struttura

```
struct point pt;
struct point {double x, double y} z,w;
struct {double x, double y},z, w;
struct point pt={2.5,2.5};
```

Vettori di strutture:

Una volta definita una struttura la si può utilizzare per costruire variabili di tipo più complicato come ad esempio *arrays* di strutture

```
struct point pv1[10];
struct point pv2[]={{0,0},{1,1}}
```

I campi di una struttura

- non possono coesistere piú campi con lo stesso nome all'interno di una struttura

- i nomi dei campi possono coincidere con nomi di variabili o funzioni, ad esempio: `struct point {double x, double y},x,y,z;`

- possono essere di tipo diverso, primitivo o strutture

```
struct rect{
        struct point pt1;
        struct point pt2;
        double   area;
};
```

- un campo di una struttura non può essere del tipo struttura che si sta definendo

Accesso ai campi di una struttura

Per individuare un membro di una struttura si utilizza l'operatore *punto*, ovvero :

`nome-struttura.membro`

Esempi

```
print("%f,%f\n",pt.x,pt.y);

double dist;
dist=sqrt(pt.x*pt.x+pt.y*pt.y);
```

Operazioni sulle strutture

- È possibile copiare strutture assegnando una variabile di tipo struttura ad una struttura dello stesso tipo:

```
struct key {
        char word[100];
        int count;
};
struct key k1={‘‘double’’,0};
struct key k2, k3=k1;
k2=k1;
```

L'assegnamento di una variabile struttura avviene per copia membro a membro pertanto nell'esempio precedente equivale a:

```
for(i=0;i<100; i++) k2.word[i]=k1.word[i];
k2.count=k1.count;
```

- Si può manipolare una struttura tramite l'accesso ai suoi membri.

- Applicando l'operatore & ad una variabile struttura si può ottenere il suo indirizzo.

- Le strutture NON possono essere confrontate, ad esempio non é legale scrivere:

```
struct point p1, p2;
if (p1 == p2) ...
```

- Si può determinare la dimensione di una struttura con
`sizeof(struct point)`

Le funzioni possono accettare come parametri e restituire come valori di ritorno variabili di tipo struttura. Al contrario di quanto accade per gli arrays il passaggio di una variabile struttura avviene per valore.

```
 struct point makepoint( double x, double y)
  {
    struct point temp;

    temp.x=x;
    temp.y=y;
    return temp
   }
struct point addpoint( struct point p1,
                       struct point p2)
 {
  p1.x+=p2.x;
  p1.y+=p2.y;
  return p1;
 }

 struct point p1,p2,ps;
 p1=makepoint( 2.5, 3.75);
 p2=makepoint(1.35 , 0.5);
 ps=addpoint(p1,p2);
```

È possibile passare *per indirizzo* una variabile struttura ad una funzione attraverso *puntatori a strutture*.

Spesso questa è una soluzione migliore in quanto quando si passa una variabile struttura a una funzione la creazione della copia locale di ogni membro della variabile stessa, può risultare molto onerosa in termini di tempo e spazio.

Puntatori a strutture

```
struct point origin={0,0},*pp;

pp=&origin;
printf(‘‘origine:(%f,%f)\n’’,(*pp).x,(*pp).y);
```

Attenzione

$$*pp.x = *(pp.x) \neq (*pp).x$$

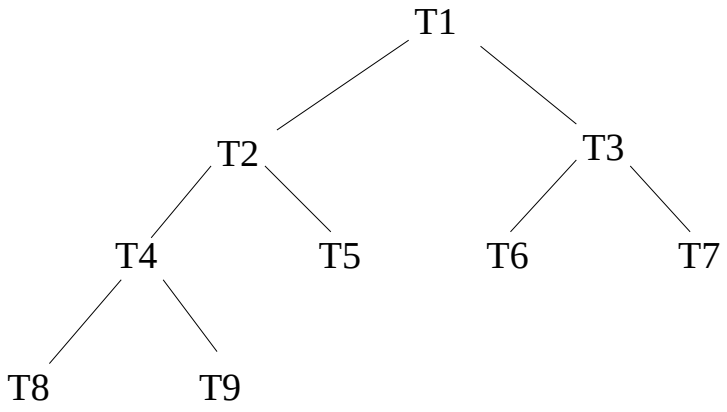
$$(*pp).x{=}pp\text{->}x$$

Siano:
`struct rect r, *rp=&r;`
le seguenti espressioni sono equivalenti:

```
r.pt1.x      rp->pt1.x
```

```
struct{
  int len;
  char *str;
} *p;

++p->len   = ++(p->len)  'incrementa len'
(++p)->len               'incrementa p e
                          poi accede a len'
(p++)->len = p++->len    'accede a len e
                          poi incrementa p'
```

Dichiarare strutture in modo ricorsivo:



triangolo T_i : $\begin{cases} \text{coordinate dei vertici} \\ \text{contatore triangolo i} \\ \text{puntatore figlio sinistro} \\ \text{puntatore figlio destro} \end{cases}$

```
struct tnode {
  struct point vertex[3];
  int          index;
  struct tnode *left;
  struct tnode * right;
};
```

Strutture che si riferiscono l'un l'altra:

```
struct t {
 ...
 struct s *p;
};

struct s {
 ...
 struct t *q;
};
```

Esempi in Alberta:

```
 struct mesh
{ ....
  struct dof_admin **dof_admin;
 ...
};

struct dof_admin
{
 struct mesh *mesh;
 ...
};
```

Typedef

Attraverso il comando `typedef` il C permette di definire nuovi nomi di tipi di dato che possono essere usati come sinonimi:

Esempio:

```
struct point { double x; double y;};
typedef struct point POINT;

POINT pt;
```

Altri esempi in Alberta:

```
typedef double  REAL;
typedef REAL    REAL_D[DIM_OF_WORLD];
typedef struct mesh  MESH;
```