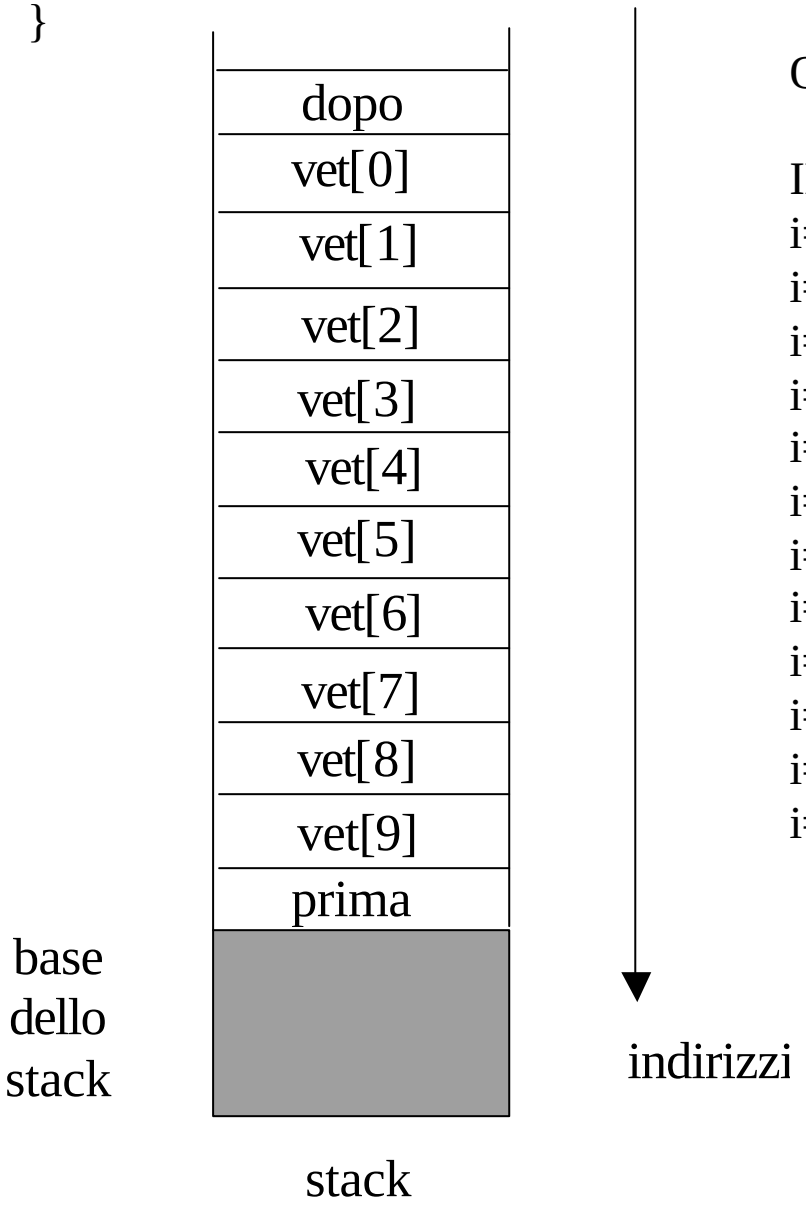
- 7° Modulo 2^parte a
  - Funzioni
    - Passaggio dei parametri
    - Relazione coi puntatori

http://www.elet.polimi.it/upload/martucci/index.html

ESEMPIO DI ERRORE di SCONFINAMENTO

```
void main(void)
{
        int prima=0;
        int vet[10];
        int dopo=0;
        printf("INIZIO: prima=%d  dopo=%d\n", prima, dopo );
        for ( i=-1; i<=10; i++) {
                vet[i] = 99;
                printf("i=%d   prima=%d  dopo=%d\n", i, prima, dopo );
        }
}
```



Output del programma

```
INIZIO: prima=0  dopo=0
i=-1   prima=0  dopo=99
i=0   prima=0  dopo=99
i=1   prima=0  dopo=99
i=2   prima=0  dopo=99
i=3   prima=0  dopo=99
i=4   prima=0  dopo=99
i=5   prima=0  dopo=99
i=6   prima=0  dopo=99
i=7   prima=0  dopo=99
i=8   prima=0  dopo=99
i=9   prima=0  dopo=99
i=10   prima=99  dopo=99
```

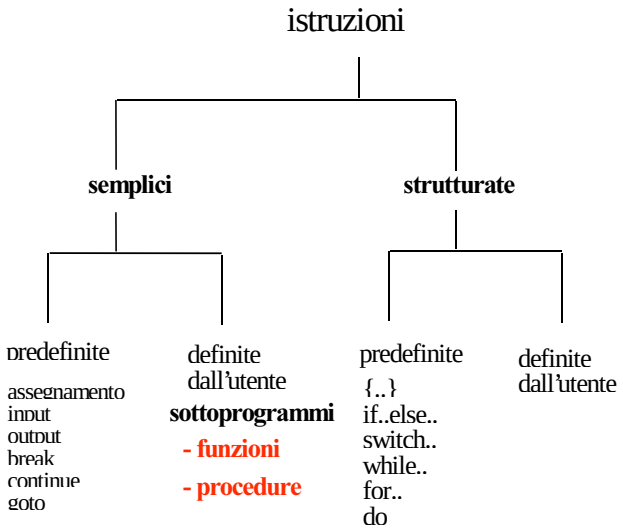
Struttura di un Programma C

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

- **Direttive per il preprocessore**
- **Definizione di tipi**
- **Prototipi di funzioni**, con dichiarazione dei tipi delle funzioni e dei parametri)
- **Dichiarazione delle Variabili Globali**
- **Dichiarazione Funzioni**, dove ogni dichiarazione di una funzione ha la forma:
  Tipo NomeFunzione(Parametri)
  {
  Dichiarazione Variabili Locali
  Istruzioni C
  }

```
#include <stdio.h>

typedef struct point {
            int   x;    int   y;
            };
int       f1(void);
void    f2(int , double );


int   sum;


void main(  )
{
      int j;
      double g=0.0;
      for(j=0;j<2;j++)
            f2(j,g);

}

void f2(int i, double g)
{
      sum = sum + g*i;
}
```

Sottoprogrammi: Funzioni e Procedure



I linguaggi di alto livello permettono di definire istruzioni non primitive per risolvere parti specifiche di un problema: i **sottoprogrammi** (funzioni e procedure).

Funzioni e Procedure

Ad esempio: Ordinamento di un insieme

```
#include <stdio.h>
#define dim 10

main()
{int V[dim], i,j, max, tmp, quanti;

/* lettura dei dati */
for (i=0; i<dim; i++)
  { printf("valore n. %d: ",i);
    scanf("%d", &V[i]);
   }
/*ordinamento */
for(i=0; i<dim; i++)
  {quanti=dim-i;
   max=quanti-1;
   for( j=0; j<quanti; j++)
      if (V[j]>V[max])
         max=j;
   if (max<quanti-1)
   {  tmp=V[quanti-1];
      V[quanti-1]=V[max];
      V[max]=tmp;
   }
  }
/*stampa */
for(i=0; i<dim; i++)
  printf("Valore di V[%d]=%d\n", i, V[i]);
}
```

- Potrebbe essere conveniente scrivere lo stesso algoritmo in modo piu‘ **astratto**:

```
#include <stdio.h>
#define dim 10

main()
{
int V[dim];

/* lettura dei dati */
leggi(V, dim);

/*ordinamento */
ordina(V, dim);

/*stampa */
stampa(V,dim);
}
```

- + **`leggi(), ordina(), stampa()`** sono ***sottoprogrammi***: il main "chiama" leggi, ordina e stampa.

Vantaggi:

sintesi
leggibilita‘
possibilita‘ di riutilizzo del codice

Sottoprogrammi: *funzioni* e *procedure*

- Rappresentano nuove istruzioni che agiscono sui dati utilizzati dal programma, "nascondendo" la sequenza delle operazioni effettivamente eseguite dalla macchina.

- Vengono realizzate mediante la definizione di unita‘ di programma (***sottoprogrammi***) distinte dal programma principale (*main*).

➳ **D'ora in poi:** il programma e‘ una **collezione di unita` di programma** (tra le quali compare l’unita‘ *main*)

Tutti i linguaggi di alto livello offrono la possibilita‘ di utilizzare funzioni e/o procedure.

Cio‘ e‘ reso possibile da:

- costrutti per la **definizione** di sottoprogrammi
- meccanismi per l’**utilizzo** di sottoprogrammi (meccanismi di ***chiamata***)

Argomenti per il main

Come comunicare con il programma

main(int argc, char** argv)

- The syntax char** argv declares argv to be a pointer to a pointer to a character
- that is, a pointer to a character array (a character string)
- in other words, an array of character strings.
- You could also write this as
  - char* argv[].

Uso di argv

- argv contains
  - *all* the information on the command line when you entered the command
  - strings are delineated by whitespace
  - *including the command itself*
- arguments, even the numeric ones, are all *strings*
- It is the programmer's job to decode them and decide what to do with them

a.out -i 2 -g -x 3 4

- argc = 7
- argv[0] = "a.out"
- argv[1] = "-i"
- argv[2] = "2"
- argv[3] = "-g"
- argv[4] = "-x"
- argv[5] = "3"
- argv[6] = "4"

prints out its own name and arguments

```
#include < stdio.h>
main(int argc, char** argv)
{ int i;
   printf("argc = %d\n", argc);
   for (i = 0; i < argc; i++)
   printf("argv[%d]=\”%s\”\n",i, argv[i]);
}

…
Argv[3]=“ABC”
…
```

Funzioni e Procedure

Definizione:

Nella fase di **definizione** di un sottoprogramma (funzione o procedura) si stabilisce:

- un **identificatore** del sottoprogramma
- si esplicita il **corpo** del sottoprogramma (cioe‘, l’insieme di istruzioni che verra‘ eseguito ogni volta che il sotto-programma verra‘ *chiamato*);
- si stabiliscono le **modalita` di comunicazione** tra l’unita‘ di programma che usa il sottoprogramma ed il sottoprogramma stesso (definizione dei **parametri formali**).

Utilizzo di funzioni/procedure (*chiamata*):

- Per chiamare un sottoprogramma (cioe‘, per richiedere l’esecuzione del suo corpo), si utilizza l’identificatore assegnato al sottoprogramma in fase di definizione (***chiamata*** o invocazione del sottoprogramma).

Meccanismo di Chiamata

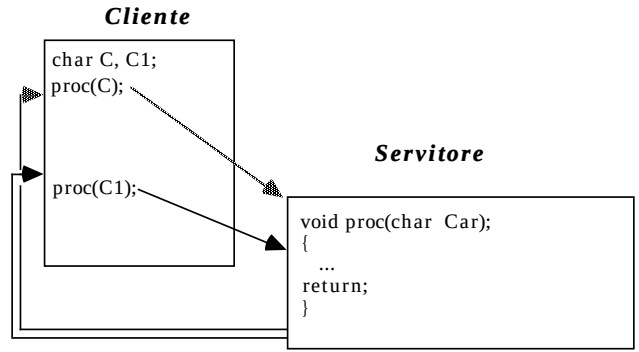
Quando si verifica una chiamata a sottoprogramma, si possono individuare due entita‘:

- l’unita‘ di programma **chiamante**;
- l’unita‘ di programma **chiamata** (il sotto-programma).

Quando avviene la chiamata, l’esecuzione dell'unita` di programma "chiamante" (quella, cioe`, che contiene l'invocazione) viene **sospesa**, ed il controllo passa al sottoprogramma chiamato (che eseguira` le istruzioni contenute nel corpo).

L'unita` chiamante funge da **cliente** dell'unita` chiamata (che svolge il ruolo di **servitore**).

Modello Cliente-Servitore



Parametri

I ***parametri*** costituiscono il mezzo di comunicazione tra unita‘ chiamante ed unita‘ chiamata.
Supportano lo scambio di informazioni tra chiamante e sottoprogramma.

parametri formali: sono quelli specificati nella definizione del sotto-programma. Sono in numero prefissato e ad ognuno di essi viene associato un tipo. Le istruzioni del corpo del sottoprogramma utilizzano i parametri formali.

parametri attuali: sono i valori effettivamente forniti dall'unita‘ chiamante al sottoprogramma all'atto della chiamata.

Parametri

- Parametri ***attuali*** (specificati nella chiamata) e ***formali*** (specificati nella definizione) devono corrispondersi in ***numero***, ***posizione*** e ***tipo***.

- All'atto della chiamata avviene il ***legame dei parametri***, cioe‘ ai parametri formali vengono associati i parametri attuali.

Come avviene l'associazione tra parametri attuali e parametri formali ?

Esistono, in generale, varie forme di legame. Ad esempio:
- legame per **valore**;
- legame per **indirizzo**;

Il significato delle due tecniche di legame dei parametri verra‘ spiegato piu‘ avanti.

Esempio:

```
int maggioredi100 (int a) /*intest. */
{/*parte dichiarazioni: */
  const int C=100;

  /* parte istruzioni: */
  if (a>C) return 1;
  else return 0;
}
```

Esempio:

```
#define N 100

typedef   char vettore[N];

int minimo (vettore vet)
{
int i, v, min;/* def. locali a minimo */
  for (min=vet[0], i=1; i<N; i++)
    {  v=vet[i];
       if (v<min) min=v;
    }
  return min;
}
```

- i, v, min sono ***variabili locali***:
  - **tempo di vita**: esistono solo durante l'esecuzione della funzione minimo
  - **visibilita`**: sono visibili (cioe' utilizzabili) soltanto all'interno della funzione minimo.

Esempio:

```
int   read_int () /* intest. */
{
  int a
  scanf("%d", &a);
  return a;
}
```

Possono esserci ***piu` istruzioni return***:

```
int max (int a, int b) /*intest.*/
{
  if (a>b)    return a;
  else   return b;
}
```

o ***nessuna***:

```
int   print_int (int a) /* intestazione */
{
  printf("%d", a);
}
```

- In questo caso, il sottoprogramma termina in corrispondenza del simbolo } ed il valore restituito e' ***indefinito.***

Esempio:

```
/* funzione elevamento a potenza */

long power (int base, int n)
{
 int i;
 long p=1;

 for (i=1;i<=n;++i)
       p *= base; /* p = p*base */
 return p; /* ritorna il risultato */
}
```

Funzioni in C

Chiamata di funzioni:

In generale, la chiamata di una funzione compare all'interno di una espressione secondo la sintassi:

…nomefunzione(<lista parametri attuali>)…

Ad esempio:

```
main()
{
int z, x=2;
...
z=power(x,2)+power(x,3);
x=max(power(z,2), 30);
printf("%d\n", x);
}
```

Realizzazione delle Procedure in C

Una funzione puo' anche avere nessun valore (void) come risultato:

void	insieme vuoto di valori (dominio vuoto)
void fun(...)	funzione che non restituisce alcun valore

➳ **In questo modo si realizza in C il concetto di** procedura

Esempio:

```
void print_int(int a)
{
  printf("%d", a);
}
```

➳ Poiche' una procedura non restituisce alcun valore, non e' necessario prevedere l'istruzione di **return** all'interno del corpo; se si utilizza, **non si deve specificare alcun argomento**:

```
return;
```

Uso:

La procedura e' l'astrazione del concetto di istruzione:

```
main()
{ int X;
  scanf("%d", &X);
  print_int(X):
}
```

Arguments to Functions

- **C passes arguments to functions by value**

  passes copies of value of the argument (call by value)

- **No direct way for called function to change value of argument**

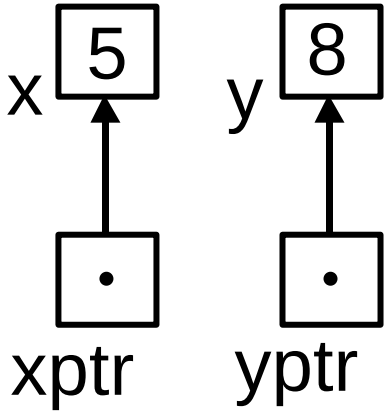
  so we "simulate" call by reference by passing address of variable; that is, a pointer!

```
swap( a, b );       /* function call */
      :
void swap( int x, int y)  /* won't work */
{
      int temp;
      temp = x;
      x = y;
      y = temp;
}
```

Arguments to Functions

- **previous code only swapped copies of arguments!**
- **To “reach” actual arguments, address is needed.**

```
swap( &x, &y );
      :
void swap( int *xptr, int *yptr )
{
      int temp;

      temp = *xptr;
      *xptr = *yptr;
      *yptr = temp;
}
```



Modalità di passaggio degli Argomenti delle Funzioni: Strutture

I dati di tipo **struct** possono venire passati alle funzioni sia per valore sia per puntatore, esplicitamente.

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

/* copia i valori di p1 nella struttura puntata da pp2 */
void copia_e_modifica_struct( struct point p1, struct point *pp2  )
{
        pp2->x  = p1.x  + 10;              /* modifiche permanenti */
        pp2->y  = p1.y   + 10;   p1.y=0; /* nessuna modifica sul dato*/
}

void main(void)
{
        struct point p1 = {   14 , 27 };
        struct point p2;
        copia_e_modifica_struct( p1, & p2  );
}
```

OSS: Se la struttura da passare è molto grande, è bene passarla per puntatore, per non sovraccaricare lo stack, soprattutto quando le chiamate sono molto annidate.

With Functions

- **By passing members**

  myfunct1(struc1.member1);

  myfunct1(aptr->member1);

  myfunct1((*aptr).member1);

- **By passing structure**

  myfunct2(struc1);

- **Passed call by value - cannot modify**

- **For call by reference, use address operator**

  myfunct3(&struc1.member1);

  myfunct4(&struc1);

Modalità di passaggio degli Argomenti delle Funzioni: Array

I dati di tipo array (vettori, matrici, array di dimensioni maggiori) char, int, long float, double e puntatori), che vengono passati come argomenti ad una funzione, **in C vengono passati per indirizzo, nel senso che "passando l'array per nome si passa l'indirizzo in cui comincia l'array".**

----- Vettori -----

Ciò significa che **quando, all'atto della chiamata, passiamo ad una funzione il nome di un vettore, passiamo l'indirizzo del primo elemento del vettore**, e non tutti i dati del vettore (i 100 interi dell' esempio)

```
void main(void)
{
        int vet[100];
        modifica_vet (vet , 100 );
}
```

di conseguenza la definizione della funzione dovrà contenere come argomento formale il puntatore al tipo di dati del vettore (un puntatore ad int nell'esempio)

```
void modifica_vet( int *v , int size  )
{
        v[0] = 137;      /* modifica conservata fuori dalla funzione */
        v = NULL;        /* annullo l'indirizzo in v, ma questa modifica
                            NON viene mantenuta fuori dalla funzione */
}
```

In tal modo se il vettore passato come argomento viene modificato dalla funzione, cioè **se la funzione modifica il contenuto degli elementi del vettore, queste modifiche sono permanenti, cioè restano nel vettore anche dopo che la funzione è terminata.**

----- Vettori , notazione alternativa ma equivalente-----

Quando l'argomento passato è un vettore, nella definizione della funzione l'argomento può essere indicato o come un puntatore oppure in un modo alternativo, come segue:

```
void modifica_vet( int v[]  , int size  )
{
        v[0] = 137;     /* modifica conservata fuori dalla funzione */
        v = NULL;       /* annullo l'indirizzo in v, ma questa modifica
                           NON viene mantenuta fuori dalla funzione */
}
```

Questa notazione **int v[]** vuole indicare che v è un vettore di interi di dimensione sconosciuta (non si sa di quanti interi è composto il vettore).

Comunque le due notazioni int *v e int v[] sono assolutamente equivalenti, entrambi i parametri vengono trattati come puntatori.

Passing to Function

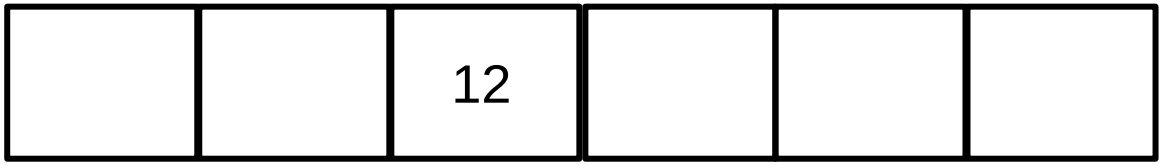
- **Passing array elements**

  call by value

  argument is array name with [ ] and index

  parameter is normal data type

int status[6];



status

print_sts(status[2]); function call

:

void print_sts(int passed_status) called function header

{

: passed_status gets value of 12 copied to it

PASSAGGIO DEI PARAMETRI: GLI ARRAY *completi!!!*

- un parametro formale array è visto come un puntatore fisso
- nell'istruzione di chiamata il parametro attuale è il nome dell'array (quindi un indirizzo)
- all'atto della chiamata il valore del nome dell'array è copiato nel nome del parametro formale array e quindi entrambi referenziano la stessa cella di memoria (**il passaggio di array è sempre per indirizzo**)

CHIAMATA E AMBIENTE

Nel chiamante

```
int        v1[N];
void       Ordina(int vettore[N]);        PROTOTIPO

Ordina (v1);                              CHIAMATA
```

In Ordina

```
void       Ordina(int vettore[N])         DEFINIZIONE
.....................

.......vettore[i]....
```

Passing to Function

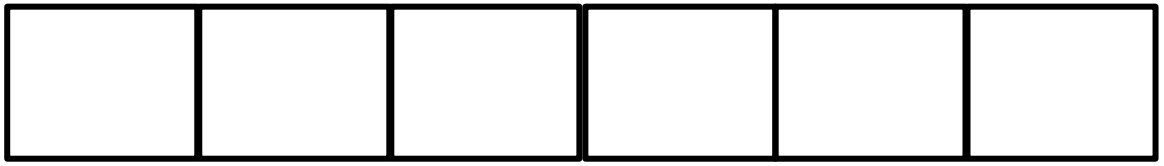
- **Passing arrays**

  call by reference

  argument is array name with no [ ]

  parameter has empty 1st [ ]

int status[6];



status

printf_sts(status);

array name is passed - an address!

•
•
•

void print_sts(int passed_array[ ])

parameter is an array of same type but called function does not know size; sizeof() will not help! (should pass as an additional parameter)

DIMENSIONI DI UN ARRAY E CALCOLO DEGLI INDIRIZZI

Accesso agli elementi di un array passato come parametro in una funzione:

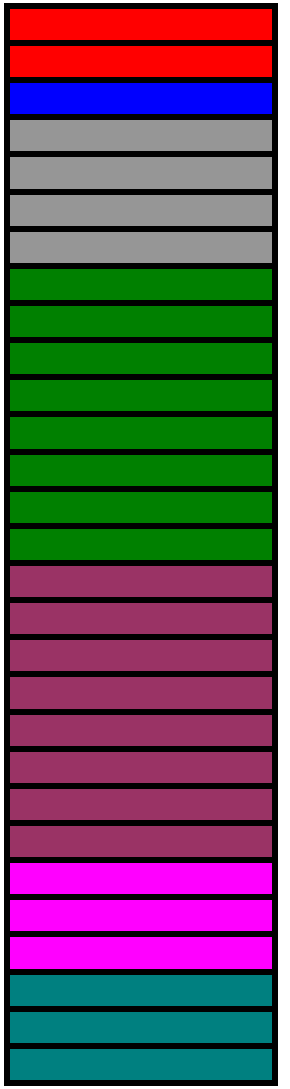
per consentire il **corretto calcolo degli indirizzi** degli elementi è necessario specificare nella testata della funzione tutte le dimensioni dell'array tranne al più la prima (quella più a sinistra). .....per calcolare l'indice r*NC+c

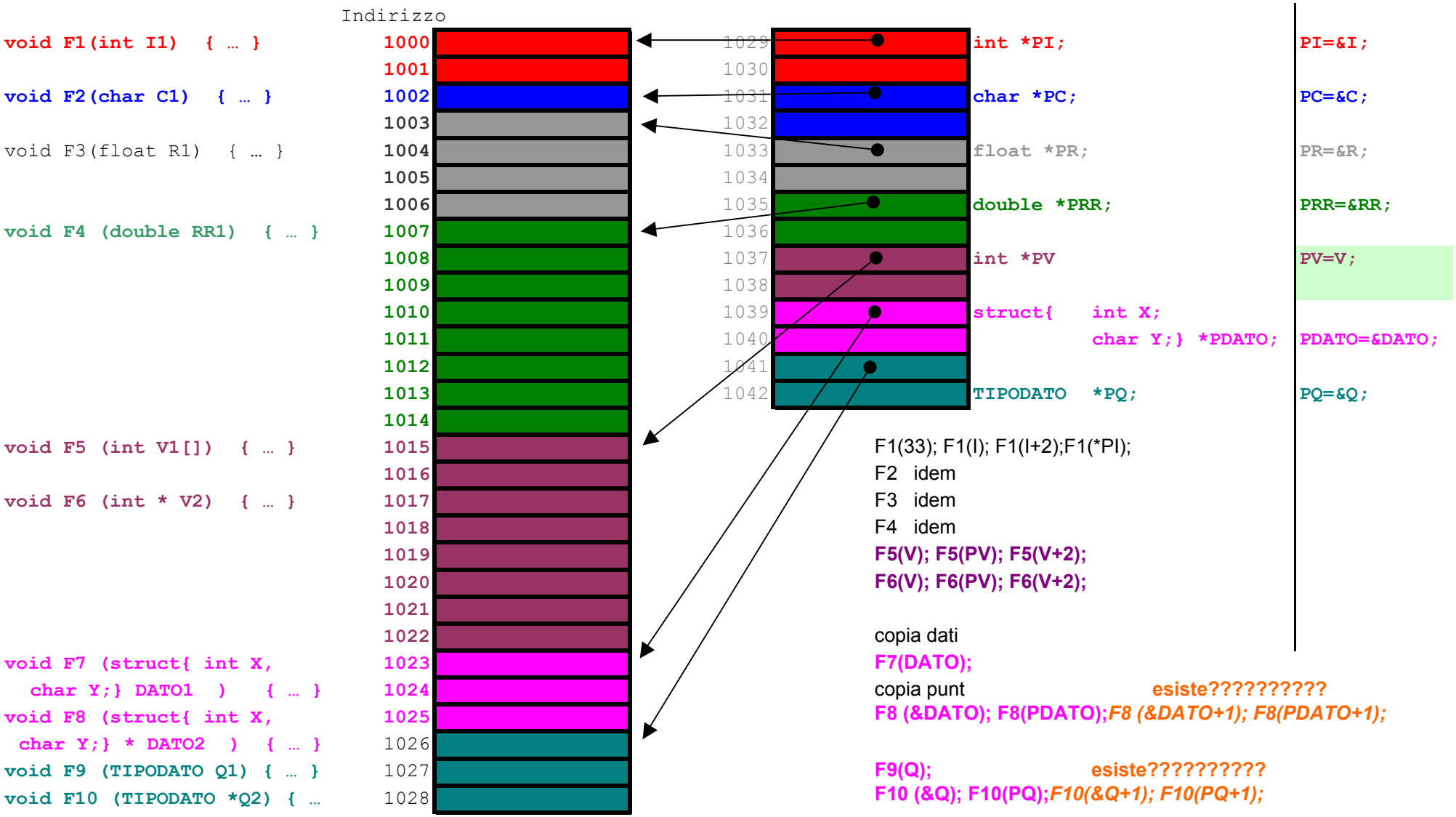
«EQUIVALENZA» TRA ARRAY E PUNTATORI: ARRAY MONODIMENSIONALI E PUNTATORI

accesso agli elementi

```
*(v+i)  v[i]
```

```
int I;                                  void F1(int I1)   { … }
char C;                                 void F2(char C1)  { … }
float R;                                void F3(float R1)  { … }
double RR;                              void F4 (double RR1)  { … }

int V[4]        V[0]   V V+0            void F5 (int V1[])  { … }
                V[1]     V+1            void F6 (int * V2)  { … }
                V[2]     V+2
                V[3]     V+3

struct{  int X,                         void F7 (struct{ int X,
         char Y;} DATO;                   char Y;} DATO1  )   { … }
typedef struct{ int A1;                 void F8 (struct{ int X,
                char A2;} TIPODATO;       char Y;} * DATO2  )  { … }
TIPODATO Q;                             void F9 (TIPODATO Q1) { … }
                                        void F10 (TIPODATO *Q2) { …
```




Indirizzo
void F1(int I1) { … }
void F2(char C1) { … }
void F3(float R1) { … }
void F4 (double RR1) { … }
void F5 (int V1[]) { … }
void F6 (int * V2) { … }
void F7 (struct{ int X, char Y;} DATO1) { … }
void F8 (struct{ int X, char Y;} * DATO2) { … }
void F9 (TIPODATO Q1) { … }
void F10 (TIPODATO *Q2) { …
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
int *PI;
char *PC;
float *PR;
double *PRR;
int *PV
struct{ int X;
char Y;} *PDATO;
TIPODATO *PQ;
PI=&I;
PC=&C;
PR=&R;
PRR=&RR;
PV=V;
PDATO=&DATO;
PQ=&Q;
F1(33); F1(I); F1(I+2);F1(*PI);
F2 idem
F3 idem
F4 idem
F5(V); F5(PV); F5(V+2);
F6(V); F6(PV); F6(V+2);
copia dati
F7(DATO);
copia punt
esiste??????????
F8 (&DATO); F8(PDATO);F8 (&DATO+1); F8(PDATO+1);
F9(Q);
esiste??????????
F10 (&Q); F10(PQ);F10(&Q+1); F10(PQ+1);

Analogamente ai vettori, **quando, all'atto della chiamata, passiamo ad una funzione il nome di una matrice, passiamo l'indirizzo del primo elemento della matrice, quindi le modifiche sui dati della matrice vengono conservate dopo l'uscita dalla funzione.**

```
void main(void)
{    int mat[10][20];
     modifica_mat ( mat );
}
void modifica_mat( int m[][20] )
{    m[1][0] = 137;       /* modifica conservata fuori dalla funzione */
     m = NULL;            /* questa modifica  NON viene mantenuta  */
}
```

Analogamente ai vettori, la definizione della funzione dovrà contenere un puntatore ad array di 20 (= num. colonne) interi.

Notare: si passa la dimensione delle righe, per calcolare l'indice r*NC+c

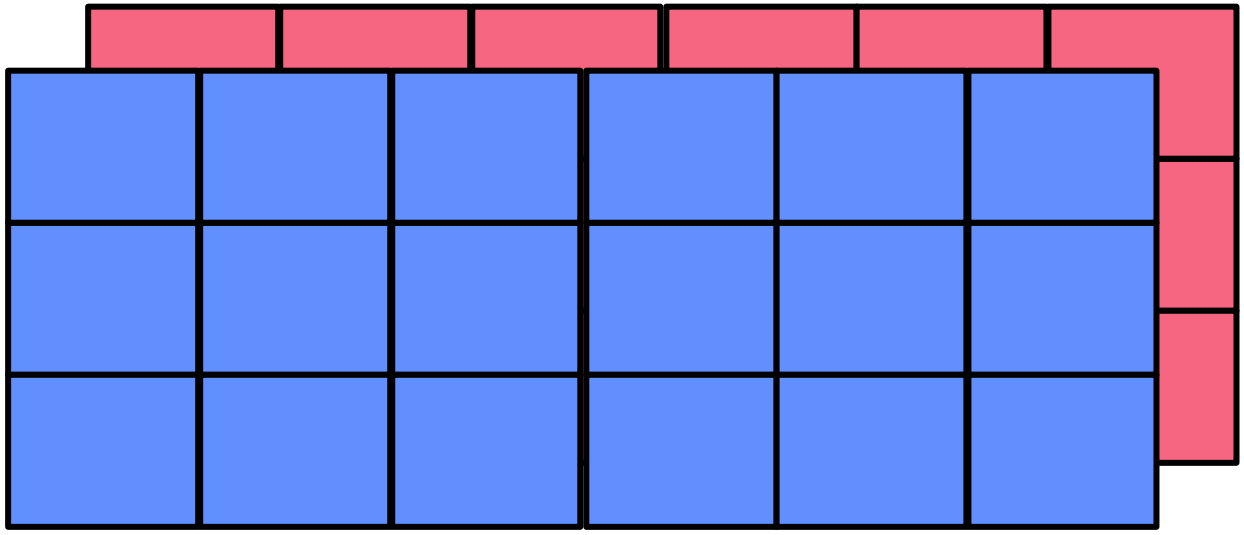
Passing Multidimensional Arrays

- **All but 1st subscript must be present**

function call -> init_array(rooms);

function header -> void init_array(int array[ ][3][6]

Highest dimension always unspecified because we may pass arrays of different sizes. Other dimensions just say how array will be manipulated.



really it’s an array of arrays all in one array !

Tipi di dato restituiti dalle Funzioni

Le funzioni possono restituire:

- dati di tipo semplice, come char, int, long, float, double, puntatori a void, o puntatori a qualche tipo di dato,
- ma anche strutture (struct) o puntatori a struct.

All'atto della chiamata, il valore restituito da una funzione:

```
double somma( double f, double g);
```

- può essere utilizzato come espressione booleana,

```
if ( somma(a,b) > 100.3 ) .....
```

- può essere utilizzato come membro di destra in un'istruzione di assegnamento,

```
f = somma(a,b);
```

- oppure può non essere considerato affatto.

```
somma(a,b);
```

Vediamo un esempio di restituzione di una struct.

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

struct point  crea_point( int x, int y  )
{
        struct point p;
        p.x = x;
        p.y = y;
        return p ;
}
void main(void)
{
        struct point p1;
        int x=21 , y = -10987;
        p1 = crea_point( x, y );
        printf ( "p1.x=%d p1.y=%d \n" , p1.x, p1.y );
}
```

Considerazioni sui Puntatori restituiti dalle Funzioni

Una funzione può restituire un puntatore ad un qualche tipo di dato, ma la correttezza nell'uso di questo puntatore si può fare, dipende da come lo spazio in memoria è allocato. Cioè:

esempio corretto: p è allocato dinamicamente quindi, anche se p è una variabile locale, lo spazio allocato sopravvive alla terminazione della funzione

```
int *alloca_vettore_1(  int size )
{
    int *p;
    p = malloc(size*sizeof(int));
    return p ;
}
```

```
void main(void)
{
    int *v;
    v = alloca_vettore ( 10 ) ;
    ...  usa v ....
}
```

---

esempio sbagliato: p è una variabile locale, lo spazio allocato non sopravvive alla terminazione della funzione

```
int *alloca_vettore_2(  int size )
{
    int p[1000];
    return p ;
}
```

```
void main(void)
{
    int *v;
    v = alloca_vettore ( 10 ) ;
    ... usa v ......
}
```

---

esempio corretto: vet_globale è una variabile globale, quindi sopravvive alla terminazione della funzione, ma cosìsia chiaro

```
int vet_globale[10000];

int *spazio_pre_allocato( void )
{
    return vet_globale ;
}
```

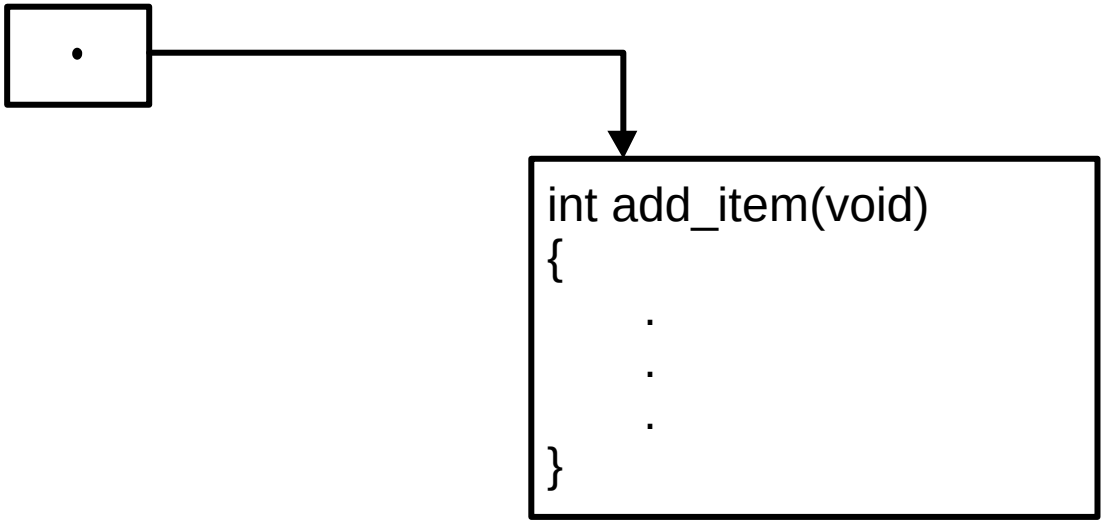
```
void main(void)
{
    int *v;
    v = spazio_pre_allocato();
    ...  usa v ....
}
```

Pointers to Functions

Function names are pointer constants

- not a variable
- can regard as address of function code
- like array name is a pointer constant

So we can have pointers to functions



Puntatori a funzione.

In C è possibile utilizzare dei puntatori a funzioni, ovvero delle variabili a cui possono essere assegnati gli indirizzi in cui risiedono le funzioni, e tramite questi puntatori a funzione, le funzioni puntate possono essere chiamate all'esecuzione.

Confrontiamo la **dichiarazione di una funzione**:

tipo_restituito **nome_funzione** (paramdef1, paramdef2, ...)

con la **dichiarazione di un puntatore a funzione**:

tipo_restituito **(* nome_ptr_a_funz)** (paramdef1, paramdef2, ...)

dove:

- paramdef1, paramdef2, ecc. sono la definizione degli argomenti da passare alla funzione all'atto della chiamata (ad es.: int i).
- tipo_restituito è il tipo del dato che viene restituito come risultato dalla funzione.

ad es, la seguente dichiarazione definisce un puntatore a funzione che punta a funzioni le quali prendono come argomenti due double, e restituiscono un double .

```
double (*ptrf) ( double g, double f);
```

Il C tratta i nomi delle funzioni come se fossero dei puntatori alle funzioni stesse.

Quindi, **quando vogliamo assegnare ad un puntatore a funzione l'indirizzo di una certa funzione dobbiamo effettuare un operazione di assegnamento del nome della funzione al nome del puntatore a funzione**

Se ad es. consideriamo la funzione dell'esempio precedente:

double somma(double a, double b);

allora **potremo assegnare la funzione somma al puntatore ptrf** così:

ptrf = somma;

Analogamente, **l'esecuzione di una funzione mediante un puntatore che la punta,viene effettuata con una chiamata in cui compare il nome del puntatore come se fosse il nome della funzione**, seguito ovviamente dai necessari parametri.

---------- esempio di uso dei puntatori a funzione ----------

per es. riprendiamo l'esempio della somma:

```
double somma( double a, double b) ;          /*  dichiarazione  o prototipo */

void main( void)
{
      double A=10 , B=29,  C;
      double (*ptrf) ( double g, double f);

      ptrf = somma;
      C = ptrf (A,B);                        /* chiamata alla funz. somma */
}

double somma( double a, double b)            /* definizione */
{     return a+b ;     }
```

Osservazione: spesso è complicato definire il tipo di dato puntatori a funzione, ed ancora di più definire funzioni che prendono in input argomenti di tipo puntatori a funzione.
In queste situazioni **è sempre bene ricorrere alla typedef per creare un tipo di dato puntatore a funzione** per funzioni che ci servono, ed utilizzare questo tipo di dato nelle altre definizioni.

---- Usare la typedef per rendere leggibile il codice C ---

Esempio:
esiste in ambiente unix (l'esempio è preso da LINUX Slackware) una funzione detta signal che puo' servire ad associare una funzione al verificarsi di un evento. Ad es. puo' servire a far eseguire una funzione allo scadere di un timer.
Il prototipo della funzione, contenuto in signal.h, e' il seguente:

```
#include <signal.h>
void (*signal(int signum, void (*handler)(int) ) )(int);          /* ????? */
```

NON E' SUBITO CHIARISSIMO COSA SIA 'STA ROBA !!!

Il significato è che
1) la funzione signal vuole come parametri un intero ***signum***, ed un puntatore ***handler*** ad una funzione che restituisce un void e che vuole come parametro un intero.
2) la funzione signal restituisce un puntatore ad una funzione che restituisce un void e che vuole come parametro un intero.

Converrebbe definire un tipo di dato come il puntatore a funzione richiesta:

```
typedef  void (*tipo_funzione) (int);
```

ed utilizzarlo per definire la signal così:

```
tipo_funzione   signal ( int signum,  tipo_funzione handler );
```

Nel man della signal e' presente questo **commento**:

> If you're confused by the prototype at the top of this manpage, it may help to see it separated out thus:
>
> typedef void (*handler_type)(int);
> handler_type signal(int signum, handler_type handler);

Pointers to Functions

Declaring pointers to functions

cannot omit parenthesis

int (*p)(); p is ptr to function that returns an int

int *p(); p is function that returns ptr to int

assign address of function as with array

```
int fname(void);
int (*fnptr)(void);
fnptr = fname;
```

fname *è un indirizzo!!!!!*

Pointers to Functions

Usage of pointers to functions

- are variables, just like other pointers
- can be used as arguments to functions to “pass” one function to another
- used to direct a sort
  - to make it data type independent
  - to allow choices of different function calls for ascending vs. descending
- used to implement “dispatch” tables
  - as might be used in a menuing system

Dissimmetria del C

```
int ar[100];
struct tag st;
```

CHIAMATE

```
func1(ar);  /* viene passato un puntatore al prmo elemeto di ar[]  */

func2 (st);  /*viene passata l'intera struttura  */
```

la stessa incoerenza si trova anche all'interno delle funzioni

DEFINIZIONI

le due seguenti sono equivalenti

```
void func1(int ar[] );

void func1 (int *ar );
```

mentre queste due basate sulle strutture NON sono equivalenti

```
void func2( struct tag st);

void func3 ( struct tag *st)
```

qui si deve passare un puntatore!!