Parametri di tipo vettore

- Il meccanismo del passaggio per valore di un indirizzo consente il passaggio di vettori come parametri di funzioni/procedure.
- Quando si passa un vettore come parametro ad una funzione, in realtà si sta passando l'indirizzo dell'elemento di indice 0.
- Il parametro formale deve essere di tipo puntatore (al tipo degli elementi del vettore)
- di solito si passa anche la dimensione del vettore in un ulteriore parametro.

Esempio:

```
void stampaVettore(int *v, int dim)
{ int i;
  for (i = 0; i < dim; i++)
    printf("v[%d]: %d\n", i, v[i]);
}

main()
{ int vet[5] = {1, 2, 3, 4, 5};
  ...
  stampaVettore(vet, 5);    ...  }
```

- Per evidenziare che il parametro formale è un vettore (ovvero l'indirizzo dell'elemento di indice 0), si può utilizzare la notazione `nome-parametro[]` invece di `*nome-parametro`.
  Esempio: `void stampa(int v[], int dim) { ... }`
- Si può anche specificare la dimensione nel parametro, ma questa viene ignorata.
  Esempio: `void stampa(int v[5], int dim) { ... }`
- Come al solito, nel prototipo della funzione il nome del parametro (vettore) può anche mancare.
  Esempio: `void stampa(int [], int);`

- Il passaggio di un vettore è un passaggio per indirizzo.

  $\Longrightarrow$ La funzione può modificare gli elementi del vettore passato.

  Esempio: Lettura di un vettore.

```
void leggiVettore(int v[], int dim)
{
  int i;
  for (i = 0; i < dim; i++)
  {
    printf("Immettere l'elemento di indice %d: ", i);
    scanf("%d", &v[i]);
  }
}
```

Esempio: Programma che legge, inverte e stampa un vettore di interi

```
#include <stdio.h>
#define LUNG 5

void leggiVettore(int [], int);
void stampaVettore(int [], int);
void invertiVettore(int [], int);

main()
{
  int vett[LUNG];

  leggiVettore(vett, LUNG);
  printf("Vettore prima dell'inversione\n");
  stampaVettore(vett, LUNG);

  invertiVettore(vett, LUNG);
  printf("Vettore dopo l'inversione\n");
  stampaVettore(vett, LUNG);
}
```

La definizione della procedura `void invertiVettore(int [], int);` è lasciata per esercizio.

Passaggio di matrici come parametri

- Quando passiamo un vettore ad una funzione, passiamo in realtà il puntatore (costante) all'elemento di indice 0.
  $\Longrightarrow$ non serve specificare la dimensione nel par. formale.
- Quando passiamo una matrice ad una funzione, per poter accedere correttamente agli elementi, la funzione deve conoscere il numero di colonne della matrice.
  $\Longrightarrow$ Non possiamo specificare il parametro nella forma `mat[][]`, come per i vettori, ma dobbiamo specificare il numero di colonne.
  Esempio: `void stampa(int mat[][5], int righe){ }`
- Il motivo è semplice: per accedere ad un generico elemento della matrice, `mat[i][j]`, la funzione deve calcolare l'indirizzo di tale elemento `mat + offset`. Per calcolare correttamente `offset` è necessario sapere quante sono le colonne.
- L'indirizzo di `mat[i][j]` è infatti:

  `mat + (i · C · sizeof(int)) + (j · sizeof(int))`

  dove `C` è il numero di colonne (gli elementi in ciascuna riga).

Riassumendo:

- per calcolare l'indirizzo dell'elemento `mat[i][j]` è necessario conoscere:
  - il valore di `mat`, ovvero l'indirizzo del primo elemento della matrice
  - l'indice di riga `i` dell'elemento
  - l'indice di colonna `j` dell'elemento
  - il numero `C` di colonne della matrice
- In generale, in un parametro di tipo array vanno specificate tutte le dimensioni, tranne eventualmente la prima.
  1. vettore: non serve specificare il numero di elementi
  2. matrice: bisogna specificare il numero di colonne, ma non serve il numero di righe

Esercizio

Definire le funzioni/procedure utilizzate nel seguente programma e completare con gli opportuni parametri attuali la chiamata di `swap` in modo che il suo effetto sia di scambiare gli elementi minimo e massimo del vettore.

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

void leggivet (int vet[], int dim);
void stampavet (int  vet[], int dim);
int indice_minimo (int vet[], int dim);
int indice_massimo (int vet[], int dim);
void swap (int *, int *);

main()
{
  int vettore[LUNG], pos_min, pos_max;

  leggivet(vettore, LUNG);
  pos_min = indice_minimo(vettore, LUNG);
  pos_max = indice_massimo(vettore, LUNG);
  swap (?, ?);   /* scambio degli elementi minimo e massimo */
  printf("Vettore dopo lo scambio dell'elemento minimo e massimo:\n");
  stampavet(vettore, LUNG);
}
```

Variabili locali

- Il blocco che costituisce il corpo di una funzione/procedura può contenere dichiarazioni di variabili.

  Esempio:

```
void leggiVettore(int v[], int dim)
{
  int i;              /* i E' UNA VARIABILE LOCALE */
  for (i = 0; i < dim; i++) { ... }
}
```

  - sono variabili proprie della funzione
  - hanno tempo di vita limitato alla durata della chiamata
  - più in generale: un identificatore dichiarato nel corpo di una funzione è detto locale alla funzione e non è visibile all'esterno della funzione (ad esempio nel `main`), ma solo nel corpo della stessa

- In realtà, ciò non è altro che un caso particolare di regole generali che governano la visibilità e il tempo di vita degli identificatori di un programma.

Struttura generale di un programma C

- parte direttiva
- parte dichiarativa globale che comprende:
  - dichiarazioni di costanti
  - dichiarazioni di tipi (li vedremo . . .)
  - dichiarazioni di variabili (variabili globali)
  - prototipi di funzioni/procedure
- il programma principale (`main`)
- le definizioni di funzioni/procedure

Esempio

```
#include <stdio.h>       /* parte direttiva */
#define LUNG 10

int i = 1;              /* variabili globali */
int j = 2;

int Q(int);             /* prototipi di funzioni e procedure */
void P(int *);

main()                 /* programma principale */
{
int x = 10;
char c = 'a';
x = Q(x);
P(&x);
}

int Q(int v) { ... }    /* definizioni di funzioni e procedure */
void P(int *z) { ... }
```

Blocchi

- il corpo di una funzione/procedura, cosi' come il corpo del programma principale, è un blocco.
- In C un blocco è costituito da
  - una parte dichiarativa (può non esserci)
  - una parte esecutiva (sequenza di istruzioni)
- Nel `main` o nel corpo delle funzioni possono comparire diversi blocchi, che possono essere
  - annidati: un blocco è una delle istruzioni di un altro blocco
  - paralleli: blocchi che fanno parte della medesima sequenza di istruzioni

```
{
   int x;
   x = 10;
   {
     int z;
     z = 20 ;
     ...
   }
   ...
}
```

```
{
   int x;
   x = 10;
   ...
}
{
   int z;
   z = 20;
   ...
}
```

- Anche la parte esecutiva del programma principale e di una funzione/procedura è un blocco
- Gli identificatori dichiarati nella parte dichiarativa di un blocco sono detti nomi locali del blocco e devono essere tutti diversi tra loro
  - nel caso di una funzione/procedura, fanno parte dei nomi locali anche gli identificatori utilizzati per i parametri formali

Esempio:

```
{
int x;   /* NO! identificatore x dichiarato   */
char x;  /* due volte nello stesso blocco     */
...
}

void p(int x, char y)
{
int x;      /* NO! identificatore x gia' usato per un parametro formale */
...
}
```

- In blocchi diversi possono essere utilizzati gli stessi identificatori

  Esempio:

```
main()
{
int x;     /* x, y: variabili locali del main */
int y;
...
   {
     char x;    /* x: variabile locale del blocco annidato */
     ...
   }
...
}

void p(int x)
{
int y;         /*x,y: variabili locali della procedura p  */
...
}
```

- Un programma C può avere una struttura molto complessa a seguito dell'uso di funzioni, procedure e blocchi.
- È necessario definire regole precise per regolamentare l'uso dei nomi utilizzati all'interno di un programma.
- A questo scopo introduciamo alcune definizioni utili.

  Ambiente globale: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa globale del programma

  Ambiente locale di una funzione: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa della funzione e nella sua intestazione

  Ambiente locale di un blocco: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa del blocco
- Quanto detto informalmente in precedenza può essere meglio precisato:

  $\Longrightarrow$ è possibile dichiarare più volte lo stesso identificatore (anche con significati diversi) purché in ambienti diversi
- Se ciò evita il proliferare di identificatori, causa il problema di stabilire il significato di un riferimento ad un identificatore in un generico punto del programma

Esempio: Riprendiamo l'esempio precedente

```
main()
{
int x;     /* x, y: variabili locali del main */
int y;
...
   {
     char x;    /* x: variabile locale del blocco annidato */
     ...
   }
...
}

void p(int x)
{
int y;         /*x,y: variabili locali della procedura p  */
...
}
```

- Se in un punto del programma viene eseguita l'istruzione `x = ...`, a quale delle tre dichiarazioni di `x` ci si riferisce?
- Dipende dal punto in cui si trova tale assegnamento e dalle regole di visibilità (o regole di scoping).

Regole di visibilità

- Gli identificatori presenti nell'ambiente globale sono visibili in tutte le funzioni e in tutti i blocchi del programma.
  Se un identificatore è definito in più punti (in blocchi e/o funzioni), la definizione valida è quella dell'ambiente più vicino al punto di utlizzo.
  N.B. Gli identificatori predefiniti del linguaggio si intendono parte dell'ambiente globale.
- Gli identificatori presenti nell'ambiente locale di una funzione sono visibili nel corpo della funzione (ivi compresi eventuali blocchi in esso contenuti).
  Se un identificatore è definito in più punti del corpo, la definizione valida è quella dell'ambiente più vicino al punto di utlizzo.
- Gli identificatori presenti nell'ambiente locale di un blocco sono visibili nella parte esecutiva del blocco (ivi compresi eventuali blocchi in essa contenuti).
  Se un identificatore è definito in più punti di un blocco, la definizione valida è quella dell'ambiente più vicino al punto di utlizzo.

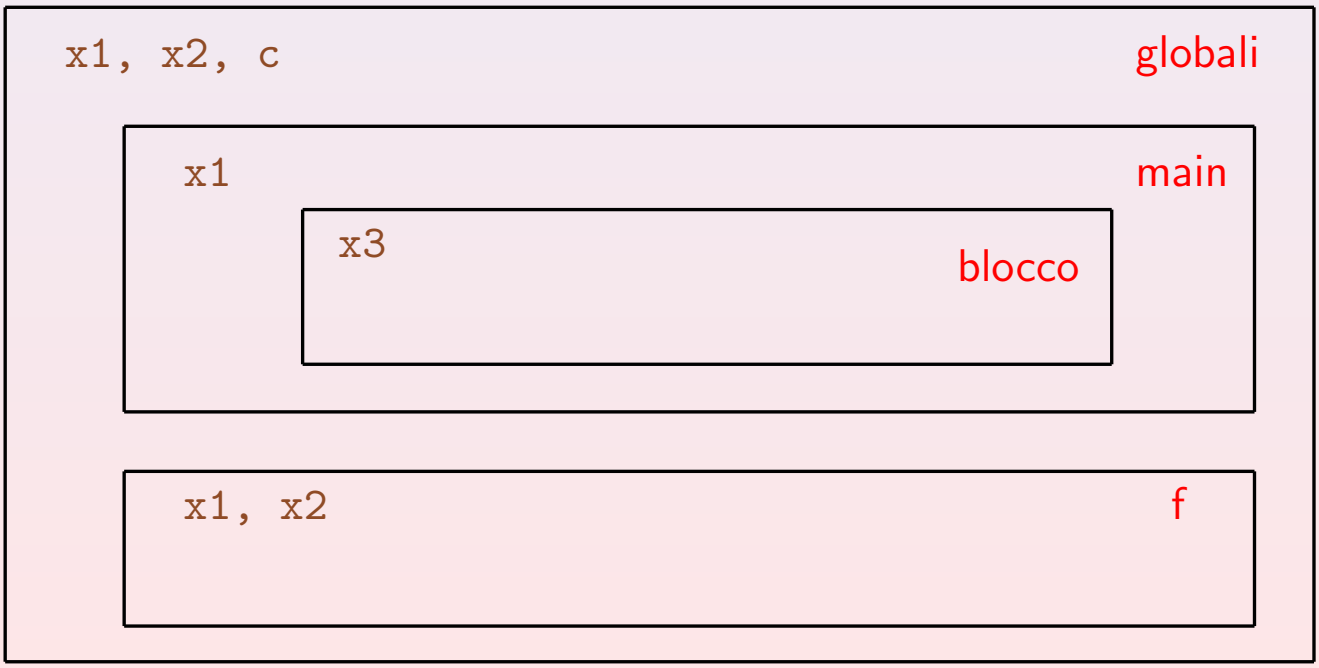
- Detto altrimenti, l'ambito di visibilità di un identificatore è determinato dalla posizione della sua dichiarazione:
  - gli identificatori dichiarati all'interno di un blocco hanno ambito di visibilità a livello di blocco
    $\Longrightarrow$ una variabile dichiarata in un blocco è visibile solo in quel blocco (compresi eventuali blocchi annidati)
  - gli identificatori dichiarati all'interno di una funzione (compresi quelli nell'intestazione) hanno ambito di visibilità a livello di funzione
    $\Longrightarrow$ una variabile dichiarata in una funzione è visibile solo nel corpo della funzione (compresi eventuali blocchi annidati)
  - gli identificatori dichiarati all'esterno delle funzioni e del main hanno ambito di visibilità a livello di programma
    $\Longrightarrow$ una variabile globale è visibile ovunque nel programma

Esempio:

```
int x1=10, x2=20;
char c='a';

int f(int);

main()
{
int x1=30;    /* nasconde la variabile globale x1 */
x2 = x1+x2;   /* x1 e' quella locale, x2 e' globale */
printf("x1=%d   x2=%d\n", x1, x2);   /* stampa x1=30  x2=50  */
  { int x3=50;
    x1=f(x3); /* x1 e' quella locale al primo blocco */
    printf("x1=%d   x2=%d\n", x1, x2);   /* stampa x1=150  x2=50 */
  }
}

int f(int x1)  /* nasconde la variabile globale x1 */
{ int x2;       /* nasconde la variabile globale x2 */
  x2 = x1 + 100;  /* x1 e' il parametro formale, x2 la var. locale */
  return x2;
}
```

Rappresentazione Grafica: Modello a contorni

- Si rappresenta ogni ambiente mediante un rettangolo con gli identificatori in esso contenuti.



Durata delle variabili

- Una variabile ha un suo tempo di vita.
  - viene creata (ovvero ad essa viene riservata uno spazio di memoria)
  - viene (o può essere) distrutta (ovvero viene rilasciato il corrispondente spazio di memoria).
- Si distinguono due classi di variabili:
  - variabili automatiche: vengono create ogni volta che si entra nel loro ambiente di visibilità e vengono distrutte all'uscita di tale ambiente
    - es. variabili locali di un blocco: vengono create all'ingresso del blocco { distrutte all'uscita dal blocco }
    - es. variabili locali di una funzione: vengono create al momento della chiamata e distrutte all'uscita
  - variabili statiche: vengono create una sola volta e vengono distrutte solo al termine dell'esecuzione del programma (non ne faremo uso ...)
- N.B. nel caso di funzioni/blocchi eseguiti più volte (es. funzione chiamata in punti diversi, blocco all'interno di un ciclo):

  le variabili automatiche corrispondenti possono essere associate di volta in volta a locazioni di memoria diverse, quindi
  il loro valore non persiste tra una esecuzione e la successiva

Gestione della memoria a tempo di esecuzione (run-time)

- Il codice macchina e i dati risiedono entrambi in memoria, ma in zone separate:
  - la memoria per il codice macchina è fissata a tempo di compilazione
  - la memoria per i dati (in particolare per le variabili automatiche) cresce e decresce dinamicamente durante l'esecuzione: viene gestita a pila
- Una pila (o stack) è una struttura dati con accesso LIFO: Last In First Out = l'ultimo entrato è il primo ad uscire (es.: pila di piatti da lavare).
- Il sistema gestisce in memoria la pila dei record di attivazione (RDA)
  - per ogni chiamata di funzione viene creato un nuovo RDA in cima alla pila
  - al termine della chiamata della funzione il RDA viene rimosso dalla pila
- Ogni RDA contiene:
  - le locazioni di memoria per i parametri formali (se presenti)
  - le locazioni di memoria per le variabili locali (se presenti)
  - altre informazioni che non analizziamo
- Anche gli ambienti locali dei blocchi vengono allocati/deallocati sulla pila.

Esempio:

```
int f(int);
main()
{
  int x, y, z;
  x=10;
  y=20;          /* blocco principale */                          PUNTO 1
  z = f(x);      /* prima chiamata di f */                        PUNTO 2
    {
    int x=50;    /* uscita da f e ingresso nel blocco annidato*/  PUNTO 3
    y=f(x);      /* seconda chiamata di f */                      PUNTO 4
    z=y;         /* uscita da f */                                PUNTO 5
    }
  ...            /* uscita dal blocco */                          PUNTO 6
}
int f(int a)
{
  int z;
  z = a + 1;
  return z;
}
```

Evoluzione della pila

x	10
y	20
z	?

PUNTO 1

Evoluzione della pila

a	10
z	?

x	10
y	20
z	?

PUNTO 2

Evoluzione della pila

x	50

x	10
y	20
z	11

PUNTO 3

Evoluzione della pila

a	50
z	?

x	50

x	10
y	20
z	11

PUNTO 4

Evoluzione della pila

x	50

x	10
y	51
z	11

PUNTO 5

Evoluzione della pila

x	10
y	51
z	51

▸ PUNTO 6

Variabili statiche: un esempio d'uso

- Una variabile statica, una volta creata, rimane in vita per tutto il tempo di esecuzione del programma.
  Esempio: `f(void) { static int x; ... }`
  - la variabile viene inizializzata alla prima attivazione della funzione
  - conserva il suo valore tra attivazioni successive
  - è locale, quindi visibile solo all'interno della funzione in cui è dichiarata

  Esempio: Funzione che ritorna il numero di volte che è stata attivata.

```
int fun1(void) {
   static int conta = 0;
       /* variabile locale statica visibile solo in fun1;
          contatore del numero di attivazioni di fun1    */
   .....
   conta++;
   return conta;
   }
```