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, di fatto, 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 del vettore nel parametro formale.
- Quando passiamo una matrice a una funzione, per poter accedere correttamente agli elementi, la funzione deve conoscere il numero di colonne della matrice. Le matrici sono infatti memorizzate linearmente, una riga dopo l'altra.
  $\Longrightarrow$ 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 il numero di colonne `C`.
- L'indirizzo di `mat[i][j]` è infatti:

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

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

  Nella matrice seguente, ad es., l'indirizzo di `mat[1][0]` è `mat + (1 · 3 · sizeof(int)) + (0 · sizeof(int))`, ovvero si trova "a 4 interi da `mat`".

4	6	5
6	1	9

- 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);
}
```

Tipi user-defined

- Il C mette a disposizione un insieme di tipi di dato predefiniti (tipi built-in) e dei meccanismi per definire nuovi tipi (tipi user-defined)
- Vediamo le regole generali che governano la definizione di nuovi tipi e quindi i costrutti linguistici (costruttori) che il C mette a disposizione.
- Tutti i tipi non predefiniti utilizzati in un programma devono essere dichiarati come ogni altro elemento del programma. Una dichiarazione di tipo viene fatta di solito nella parte dichiarativa del programma.
  - parte dichiarativa globale:
    - dichiarazioni di costanti
    - dichiarazioni di tipi
    - dichiarazioni di variabili
    - prototipi di funzioni/procedure

Dichiarazione di tipo

- Una dichiarazione di tipo (type declaration) consiste nella parola chiave typedef seguita da:
  - la rappresentazione o costruzione del nuovo tipo (ovvero la specifica di come è costruito a partire dai tipi già esistenti)
  - il nome del nuovo tipo
  - il simbolo ; che chiude la dichiarazione

  Esempio: `typedef int anno;`
- Una volta definito e nominato un nuovo tipo, è possibile utilizzarlo per dichiarare nuovi oggetti (ad es. variabili) di quel tipo.

  Esempio:

```
float x;
anno a;
```

- **Nota:** In C si possono anche definire tipi senza usare `typedef`. Quest'ultima consente l'associazione di un nome (identificatore) a un nuovo tipo. Per uniformità e leggibilità del codice useremo spesso `typedef` per definire nuovi tipi.

Tipi semplici user-defined

Ridefinizione: Un nuovo tipo può essere definito rinominando un tipo già esistente (cioè creandone un alias)

typedef TipoEsistente NuovoTipo;

dove `TipoEsistente` può essere un tipo built-in o user-defined.

Esempio:

```
typedef int anno;
typedef int naturale;

typedef char carattere;
```

Enumerazione: Consente di definire un nuovo tipo enumerando i suoi valori, con la seguente sintassi

typedef enum {v1, v2, ... , vk} NuovoTipo;

Esempio:

```
typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
typedef enum {gen, feb, mar, apr, mag, giu,
              lug, ago, set, ott, nov, dic} Mese;

typedef enum {m, f} sesso;
```

- I valori elencati nella definizione di un nuovo tipo enumerato, sono identificatori che rappresentano costanti di quel tipo (esattamente come `0, 1, 2, ...` sono costanti del tipo `int`, o `'a', 'b', ...` sono costanti del tipo `char`).
- Dunque, se dichiariamo una variabile
  `Giorno g;`
  possiamo scrivere l'assegnamento
  `g = mar;`
- Le costanti dei tipi enumerati non vanno racchiuse tra virgolette o tra apici!

  N.B. Il compilatore associa ai nomi utilizzati per denotare le costanti dei tipi enumerati valori naturali progressivi.

  Esempio: il valore associato a `g` dopo l'assegnamento `g=mar` è il numero naturale (intero) `1`.

  $\Longrightarrow$ mancanza di astrazione: è possibile fare riferimento alla rappresentazione dei valori.

- La relazione tra interi e tipi enumerati consente di applicare a questi ultimi le seguenti operazioni:
  - operazioni aritmetiche: `+,-,*,/,%`
  - uguaglianza e disuguaglianza: `=, !=`
  - confronto: `<,<=,>,>=`
- Si noti che la relazione di precedenza tra i valori (che determina l'esito delle operazioni di confronto) dipende dall'ordine in cui vengono elencati i valori del tipo al momento della sua definzione.

  Esempio: Con le dichiarazioni viste in precedenza `lun < gio` è vero (un intero diverso da `0`) `apr <= feb` è falso (il valore intero `0`)
- Il C tratta questi tipi come ridefinizione di `int`

Tipi fai da te: i booleani

Soluzione 1

```
typedef int Boolean;
Boolean b; ...
```

Soluzione 2

```
#define FALSE 0;
#define TRUE 1;...
typedef int Boolean;
Boolean b;
...
```

Soluzione 3

```
typedef enum {FALSE, TRUE} Boolean;...
Boolean b;
...
```

N.B. I valori vanno elencati come sopra, rispettando la convenzione adottata dal `C`: il valore `0` rappresenta falso.

Esempio:

```
typedef enum {false, true} boolean;

boolean even (int n)
{
if (n % 2 == 0)
    return true;
else
    return false;
}


boolean implies (boolean p, boolean q)
{
    if (p)
        return q;
    else
        return true;
}
```

Esempio: Uso del costrutto `switch` con tipi enumerati

```
typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
Giorno g;
...
switch (g) {
case lun: case mar: case mer: case gio: case ven:
        printf("Giorno lavorativo");
        break;
case sab: case dom:
        printf("Week-end");
        break;
}
```

```
void stampaGiorno(Giorno g) {
switch (g) {
case lun: printf("lun");
          break;
...

case dom: printf("dom");
          break;
}
```

Tipi strutturati user-defined

- Il C non possiede tipi strutturati built-in, ma fornisce dei costruttori che permettono di definire tipi strutturati anche piuttosto complessi.
- Array e puntatori possono essere visti come costruttori di tipo (definiscono un tipo di dato non semplice a partire da tipi esistenti).

Uso di `typedef` con array e puntatori

- In generale, una dichiarazione di tipo mediante `typedef` ha la forma di una dichiarazione di variabile preceduta dalla parola chiave `typedef`, e con il nome di tipo al posto del nome della variabile.
- Nel caso di array e puntatori:

```
typedef TipoElemento TipoArray[Dimensione];
typedef TipoPuntato *TipoPuntatore;
```

Esempio:

```
typedef int ArrayDieciInteri[10];
typedef int MatriceTreXQuattro[3][4];
typedef int *PuntIntero;
ArrayDieciInteri vet;           /*  int vet[10]; */
PunIntero p;                    /*  int *p; */
MatriceTreXQuattro mat, mat1;   /*  int mat[3][4]; int mat1[3][4]; */
```

Il costruttore struct

- Una struttura è un'aggregazione di elementi che possono essere eterogenei (di tipo diverso).
  Esempio:

```
struct persona  {
      char nome[15];
      char cognome[20];
      int eta;
      sesso s;   }
```

  - la parola chiave `struct` introduce la definizione della struttura
  - `persona` è l'etichetta della struttura, attribuisce un nome alla definizione della struttura
  - `nome`, `cognome`, `eta`, `s` sono detti campi della struttura
- È anche possibile definire strutture con campi omogenei

```
struct complex  {
 double real;
 double imag;  }
```

Campi di una struttura

- devono avere nomi univoci all'interno di una struttura
- strutture diverse possono avere campi con lo stesso nome
- i nomi dei campi possono coincidere con altri nomi già utilizzati (es. per variabili o funzioni)

  Esempio:

```
int x;
struct a { char x; int y; };
struct b { int w; float x; };
```

- possono essere di tipo diverso (semplice o altre strutture)
- un campo di una struttura non può essere del tipo struttura che si sta definendo
- un campo può però essere di tipo puntatore alla struttura

  Esempio:

```
struct s { int a;
           struct s *p;  };
```

Dichiarazione di variabili di tipo struttura

- La definizione di una struttura non provoca allocazione di memoria, ma introduce un nuovo tipo di dato.
- Una volta definito un tipo struttura lo si può usare per costruire variabili di tipi più complicati: array di strutture, puntatori a strutture, ecc.

  Esempio: `struct persona tizio, docenti[10], *p;`
  - `tizio` è una variabile di tipo `struct persona`
  - `docenti` è un vettore di 10 elementi di tipo `struct persona`
  - `p` è un puntatore a una `struct persona`
  - N.B.: `persona tizio;` Errore!
- Una variabile di tipo struttura può essere dichiarata contestualmente alla definizione della struttura (si può omettere l'etichetta).

  Esempio:

```
struct studente {          |      struct  {
  char nome[20];           |        char nome[20];
  long matricola;          |        long matricola;
  struct data ddn;         |        struct data ddn;
} s1, s2;                  |      } s1, s2;
```

Uso di typedef con strutture

- L'istruzione `typedef` introduce sinonimi per tipi costruibili in C. Spesso viene usata per semplificare dichiarazioni complesse e/o per rendere più intuitivo l'uso di un tipo in una particolare accezione.

  Esempio:

```
struct data { int giorno, mese, anno; };

typedef struct data Data;
```

- L'identificatore `Data` è un sinonimo del tipo `struct data`, che può essere utilizzato nelle dichiarazioni di variabili.

```
Data d1, d2;
Data appelli[10], *pd;
```

Operazioni sulle strutture

- Si possono assegnare variabili di tipo struttura a variabili dello stesso tipo struttura. L'assegnamento di una variabile struttura a un'altra avviene attraverso la copia membro a membro. Se la struttura contiene un array come membro, anche l'array viene duplicato.

  Esempio:

```
Data d1, d2;
...
d1 = d2;
```

- Non è possibile invece effettuare il confronto tra due variabili di tipo struttura: si devono avere funzioni che confrontano le strutture

  Esempio:

```
struct data d1, d2;
if (d1 == d2) ...          Errore!
```

- L'equivalenza di tipo tra strutture è per nome.

  Esempio:

```
struct s1 { int i; };
struct s2 { int i; };
struct s1 a, b;
struct s2 c;
```

`a = b;`	OK	`a` e `b` sono dello stesso tipo
`a = c;`	Errore!	`a` e `c` non sono dello stesso tipo

- Si può ottenere l'indirizzo di una variabile di tipo struttura tramite l'operatore `&`.
- Si può rilevare la dimensione di una struttura con `sizeof`.

  Esempio: `sizeof(struct data)`
- Attenzione: non è detto che la dimensione di una struttura sia pari alla somma delle dimensioni dei singoli campi.

Accesso ai campi di una struttura

- I campi di una struttura si comportano come variabili del tipo corrispondente e si accedono per nome. L'accesso avviene tramite l'operatore punto

```
Data oggi;
oggi.giorno = 27; oggi.mese = 11; oggi.anno = 2023;
printf("%d %d %d", oggi.giorno, oggi.mese, oggi.anno);
```

- Accesso tramite un puntatore alla struttura.

```
Data oggi, *pd;
pd = &oggi;
(*pd).giorno = 27; (*pd).mese = 11; (*pd).anno = 2023;
```

N.B. Ci vogliono le **()** perché "**.**" ha priorità più alta di "*****".

- Operatore freccia: combina il dereferenziamento e l'accesso al campo della struttura.

```
pd->giorno =27; pd->mese = 11; pd->anno = 2023;
```

- N.B.: `pd->giorno` è una abbreviazione per `(*pd).giorno`.

Tra gli elementi di una struttura può essere contenuta un'altra struttura

Esempio: Accesso al campo di una struttura che è a sua volta campo di un'altra struttura: tanti punti quanti i livelli di annidamento.

```
struct dipendente
   { Persona datiDip;
     Data dataAssunzione;
     int stipendio;
   };
typedef struct dipendente Dipendente;

Dipendente dip, *p;
...
dip.dataAssunzione.giorno = 3;
dip.dataAssunzione.mese = 4;
dip.dataAssunzione.anno = 1997;
...
(p->dataAssunzione).giorno = 5;
(p->stipendio) = (p->stipendio) + 120;
```

Inizializzazione di strutture

- Può avvenire, come per i vettori, con un elenco di inizializzatori.

  Esempio: `Data oggi = { 27, 11, 2023 }`

- Se ci sono meno inizializzatori di campi della struttura, i campi rimanenti vengono inizializzati a 0 (o al valore speciale `NULL`, se il campo è un puntatore).

Passaggio di parametri di tipo struttura

- È come per i parametri di tipo semplice e **non** come per gli array:
  - il passaggio è per valore $\Longrightarrow$ viene fatta una copia dell'intera struttura dal parametro attuale a quello formale, inclusi i membri di tipo array.
  - è comunque possibile simulare il passaggio per indirizzo attraverso un puntatore

  Nota: per passare per valore ad una funzione un vettore (il vettore, non il puntatore al suo primo elemento) è sufficiente racchiuderlo in una struttura.

Esempio:

```
struct dipendente
   { Persona datiDip;
     Data dataAssunzione;
     int stipendio;
   };
typedef struct dipendente Dipendente;

void aumento(Dipendente *p, int percentuale)
{
    int incremento;
    incremento = (p -> stipendio) * percentuale / 100;
    p -> stipendio = p -> stipendio + incremento;
}
```