

# TIPI DI DATO

---

- Tipicamente un elaboratore è **capace** di trattare domini di dati di **tipi primitivi**
  - *numeri naturali, interi, reali*
  - *caratteri e stringhe di caratteri*e quasi sempre anche collezioni di oggetti, mediante la definizione di **tipi strutturati**
  - *array, strutture*
- Spesso un linguaggio di programmazione permette di **introdurre altri tipi definiti dall'utente**

# TIPI DEFINITI DALL'UTENTE

---

- In C, l'utente può introdurre *nuovi tipi* tramite una ***definizione di tipo***
- **La definizione associa a un identificatore (*nome del tipo*) un tipo di dato**
  - aumenta la leggibilità del programma
  - consente di ragionare per astrazioni
- Linguaggio C consente in particolare di:
  - **ridefinire tipi già esistenti**
  - **definire dei nuovi *tipi strutturati***
  - **definire dei nuovi *tipi enumerativi***

# TIPI RIDEFINITI

---

*Un nuovo identificatore di tipo viene dichiarato identico* a un tipo già esistente

Schema generale:

```
typedef TipoEsistente NuovoTipo;
```

Esempio

```
typedef      int MioIntero;  
MioIntero    X, Y, Z;  
int          W;
```

# DEFINIZIONE DI TIPI STRUTTURATI

---

Abbiamo visto a suo tempo come introdurre *variabili* di tipo array e struttura:

```
char msg1[20], msg2[20];  
struct persona {...} p, q;
```

Non potendo però *dare un nome* al nuovo tipo, dovevamo *ripetere la definizione* per ogni nuova variabile

- per le strutture potevamo evitare di ripetere la parte fra {...}, ma `struct persona` andava ripetuto comunque

# DEFINIZIONE DI TIPI STRUTTURATI

---

Ora siamo in grado di ***definire nuovi tipi array e struttura***:

```
typedef char string[20];  
typedef struct {...} persona;
```

Ciò consente di ***non dover più ripetere la definizione per esteso*** ogni volta che si definisce una nuova variabile:

```
string s1, s2; /* due stringhe di 20 caratteri */  
persona p1, p2; /* due strutture "persona" */
```

- per le strutture, ciò rende ***quasi sempre inutile specificare etichetta*** dopo parola chiave `struct`

# TIPI ENUMERATIVI

---

Un *tipo enumerativo* viene specificato tramite *l'elenco dei valori* che i dati di quel tipo possono assumere

**Schema generale:**

```
typedef enum {  
    a1, a2, a3, ... , aN } EnumType;
```

**Il compilatore associa a ciascun “identificativo di valore”  $a_1, \dots, a_N$  un *numero naturale* (0,1,...), che viene usato nella valutazione di espressioni che coinvolgono il nuovo tipo**

# TIPI ENUMERATIVI

---

Gli “identificativi di valore”  $a_1, \dots, a_N$  sono a tutti gli effetti delle *nuove costanti*

Esempi:

```
typedef enum {
    lu, ma, me, gi, ve, sa, dom} Giorni;
typedef enum {
    cuori, picche, quadri, fiori} Carte;

Carte    C1, C2, C3, C4, C5;
Giorni   Giorno;
if (Giorno == dom) /* giorno festivo */
else /* giorno feriale */
```

# TIPI ENUMERATIVI

---

Un “identificativo di valore” può comparire *una sola volta* nella definizione di *un solo tipo*, altrimenti si ha ambiguità

Esempio:

```
typedef enum {  
    lu, ma, me, gi, ve, sa, dom} Giorni;  
typedef enum { lu, ma, me} PrimiGiorni;
```

La definizione del secondo tipo enumerativo è **scorretta**, perché gli identificatori `lu`, `ma`, `me` sono già stati usati altrove



# TIPI ENUMERATIVI

---

**Un tipo enumerativo è *totalmente ordinato*:**  
vale l'ordine con cui gli identificativi di valore  
sono stati elencati nella definizione

Esempio:

```
typedef enum {  
    lu, ma, me, gi, ve, sa, dom} Giorni;
```

Data questa definizione,

`lu < ma`      *è vera*

`lu >= sa`     *è falsa*

in quanto `lu`  $\leftrightarrow$  0, `ma`  $\leftrightarrow$  1, `me`  $\leftrightarrow$  2, ...

## TIPI ENUMERATIVI

---

Poiché un tipo enumerativo è, *per la macchina C*, indistinguibile da un intero, è possibile, anche se sconsigliato, ***mescolare interi e tipi enumerativi***

Esempio:

```
typedef enum {  
    lu, ma, me, gi, ve, sa, dom} Giorni;  
Giorni g;  
g = 5;          /* equivale a g = sa */
```

## TIPI ENUMERATIVI

---

È anche possibile *specificare esplicitamente i valori naturali cui associare i simboli*

**a1, ..., aN**

- qui,  $lu \leftrightarrow 0$ ,  $ma \leftrightarrow 1$ ,  $me \leftrightarrow 2$ , ...

```
typedef enum {  
    lu, ma, me, gi, ve, sa, dom} Giorni;
```

- qui, invece,  $lu \leftrightarrow 1$ ,  $ma \leftrightarrow 2$ ,  $me \leftrightarrow 3$ , ...

```
typedef enum {  
    lu=1, ma, me, gi, ve, sa, dom} Giorni;
```

- qui, infine, l'associazione è data caso per caso

```
typedef enum { lu=1, ma, me=7, gi, ve,  
    sa, dom} Giorni;
```

# IL TIPO BOOLEAN

---

*Il boolean non esiste in C*, ma si può facilmente definire in termini di tipo enumerativo:

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

Di conseguenza:

`false`  $\leftrightarrow$  0, `true`  $\leftrightarrow$  1

`false` < `true`

# EQUIVALENZA

---

- La possibilità di introdurre nuovi tipi pone il problema di ***stabilire se e quanto due tipi siano compatibili fra loro***
- **Due possibili scelte:**
  - ***equivalenza strutturale***  
tipi equivalenti se ***strutturalmente identici***
  - ***equivalenza nominale***  
tipi equivalenti se ***definiti nella stessa definizione*** oppure se ***il nome dell'uno è definito espressamente come identico all'altro***

Scelta dal C

# EQUIVALENZA STRUTTURALE

---

Esempio di **equivalenza strutturale**

```
typedef int MioIntero;  
typedef int NuovoIntero;  
MioIntero A;  
NuovoIntero B;
```

**I due tipi `MioIntero` e `NuovoIntero` sono equivalenti perché *strutturalmente identici* (entrambi `int` per la macchina C)**

Quindi, `A=B` è un assegnamento lecito

# EQUIVALENZA NOMINALE

---

- Non è il caso del C, ma è il caso, per esempio, del **Pascal**
- Esempio di **equivalenza nominale**

```
type MioIntero = integer;  
type NuovoIntero = integer;  
var A: MioIntero;  
var B: NuovoIntero;
```

- I due tipi **MioIntero** e **NuovoIntero** non sono equivalenti perché definiti in una diversa definizione ( $A := B$  non è consentito)

# Sistema lineare

- Scrivere una procedura/funzione che risolva un *sistema lineare di due equazioni in due incognite*

$$\left\{ \begin{array}{l} a_1x + b_1y = c_1 \\ a_2x + b_2y = c_2 \end{array} \right.$$

- Soluzione:

$$x = (c_1b_2 - c_2b_1) / (a_1b_2 - a_2b_1) = X_N / D$$

$$y = (a_1c_2 - a_2c_1) / (a_1b_2 - a_2b_1) = Y_N / D$$

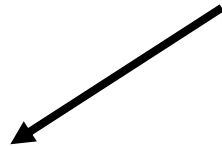


# Sistema lineare

- Seguire i passi delineati nell'esempio precedente
  - Controllo errore  $\rightarrow$  Valore di ritorno
  - Coefficienti  $\rightarrow$  Parametri per valore
  - Soluzioni  $\rightarrow$  Parametri per indirizzo
- Controllo errore
  - Ok, se  $X_N \neq 0, Y_N \neq 0, D \neq 0$
  - Impossibile, se  $X_N \neq 0, Y_N \neq 0, D = 0$
  - Indeterminato, se  $X_N = 0, Y_N = 0, D = 0$
  - $\rightarrow$  Tre possibili valori... un “enumerativo”!

# Sistema lineare

- Interfaccia



*Definisce un tipo che può assumere solo i valori specificati → i valori sono mappati su interi (da 0 in poi...)*

```
typedef enum { ok , impossibile, indeterminato }  
    TipoSistema;
```

```
TipoSistema sistema(int a1, int b1, int c1,  
    int a2, int b2, int c2,  
    float *x, float *y);
```

# Sistema Lineare

```
int main()
{
    TipoSistema tipoSistema;
    int a1, b1, c1, a2, b2, c2; float x, y;
    printf("Inserire coefficienti eq. 1: ");
    scanf("%d %d %d\n", &a1, &b1, &c1);
    printf("inserire coefficienti eq. 2: ");
    scanf("%d %d %d\n", &a2, &b2, &c2);
    tipoSistema = sistema(a1, b1, c1, a2, b2, c2, &x, &y);
    switch (tipoSistema)
    {
        case ok: printf("%f %f\n", x, y);
                break;
        case impossibile: printf("Sistema impossibile");
                break;
        case indeterminato: printf("Sistema indeterminato");
                break;
    }
}
```

# Sistema Lineare

```
TipoSistema sistema (int a1, int b1, int c1, int a2, int b2,
    int c2, float *x, float *y)
{
    int XN, YN, D;
    XN = c1*b2 - c2*b1;
    YN = a1*c2 - a2*c1;
    D = a1*b2 - a2*b1;
    if (D == 0)
    {
        if (XN == 0) return indeterminato;
        else return impossibile;
    }
    else
    {
        *x = (float) (XN) / D;
        *y = (float) (YN) / D;
        return ok;
    }
}
```