

# 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`.

⇒ 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 definizione.  
**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

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

### Soluzione 2

```
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]; */
PuntIntero 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.

**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.

**Esempio:**

<code>struct studente {</code>		<code>struct {</code>
<code>  char nome[20];</code>		<code>  char nome[20];</code>
<code>  long matricola;</code>		<code>  long matricola;</code>
<code>  struct data ddn;</code>		<code>  struct data ddn;</code>
<code>} s1, s2;</code>		<code>} s1, s2;</code>

- ▶ In questo caso si può anche **omettere l'etichetta** di struttura.

## Uso di typedef con strutture

- ▶ Attraverso `typedef` è possibile associare un nome ad un tipo definito mediante il costruttore `struct`.

### Esempio:

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

- ▶ `Data` è un **sinonimo** di `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.

## Esempio:

```
Data d1, d2;
```

```
...
```

```
d1 = d2;
```

- ▶ **Non** è possibile invece effettuare il confronto tra due variabili di tipo struttura.

## 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. L'accesso avviene tramite l'**operatore punto**

```
Data oggi;
oggi.giorno = 11; oggi.mese = 5; oggi.anno = 2009;
printf("%d %d %d", oggi.giorno, oggi.mese, oggi.anno);
```

- ▶ Accesso tramite un puntatore alla struttura.

```
Data oggi, *pd;
pd = &oggi;
(*pd).giorno = 11; (*pd).mese = 5; (*pd).anno = 2009;
```

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

- ▶ **Operatore freccia**: combina il dereferenzamento e l'accesso al campo della struttura.

```
pd->giorno =11; pd->mese = 5; pd->anno = 2009;
```

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



**Esempio:** Accesso al campo di una struttura che è a sua volta campo di un'altra struttura.

```
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 = { 11, 5, 2009 }`

- ▶ 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:
  - ▶ il passaggio è **per valore**  $\implies$  viene fatta una **copia dell'intera struttura** dal parametro attuale a quello formale
  - ▶ 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;
}
```