

## 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**

1

## 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

2

## 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;
```

3

## 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

4

## 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
```

5

## 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

6

## 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 */
```

7

## 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

8

## 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, ...

9

## 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 */
```

10

## 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;
```

11

## 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

12

## EQUIVALENZA

- La possibilità di introdurre nuovi tipi pone il problema di *stabilire se e quanto due tipi siano compatibili fra loro*
- Due possibili scelte: Scelta dal C
  - **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*

13

## 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

14

## 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)

15