

tipi definiti dall'utente

testo Cap. 9.2

1) tipo enumerazione

2) tipo struttura

essenziale: capire come si integrano nel linguaggio

1. Tipo enumerazione

esempio: i giorni della settimana:

```
enum giorno {lunedì, martedì, mercoledì,  
giovedì, venerdì, sabato, domenica};
```

TOKEN

```
giorno x;
```

```
x= martedì;
```

default: lunedì = 0, martedì =1, mercoledì=2,...

```
cout << x; // stampa 1
```

int k= domenica + 10; *// ok k=16*

promozione automatica enum → int

ma non viceversa:

giorni x=10;

richiede int → enum *//non va !*

giorni x= sabato;

x++; *// NO*

richiede enum → int e int → enum *//NO*

deve essere chiaro:

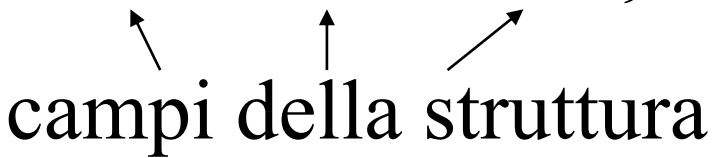
i tipi enumerazione consentono “solo” di introdurre nel programma delle costanti con nomi intuitivi (lunedì ecc.)

il loro uso è “solo” quello di rendere il programma più leggibile

non funzionano con input e output e infatti non esistono nel codice oggetto (in compilazione sono sostituite con i corrispondenti interi)

struttura

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



campi della struttura

```
data x; // dichiarazione di x senza inizializzazione
```

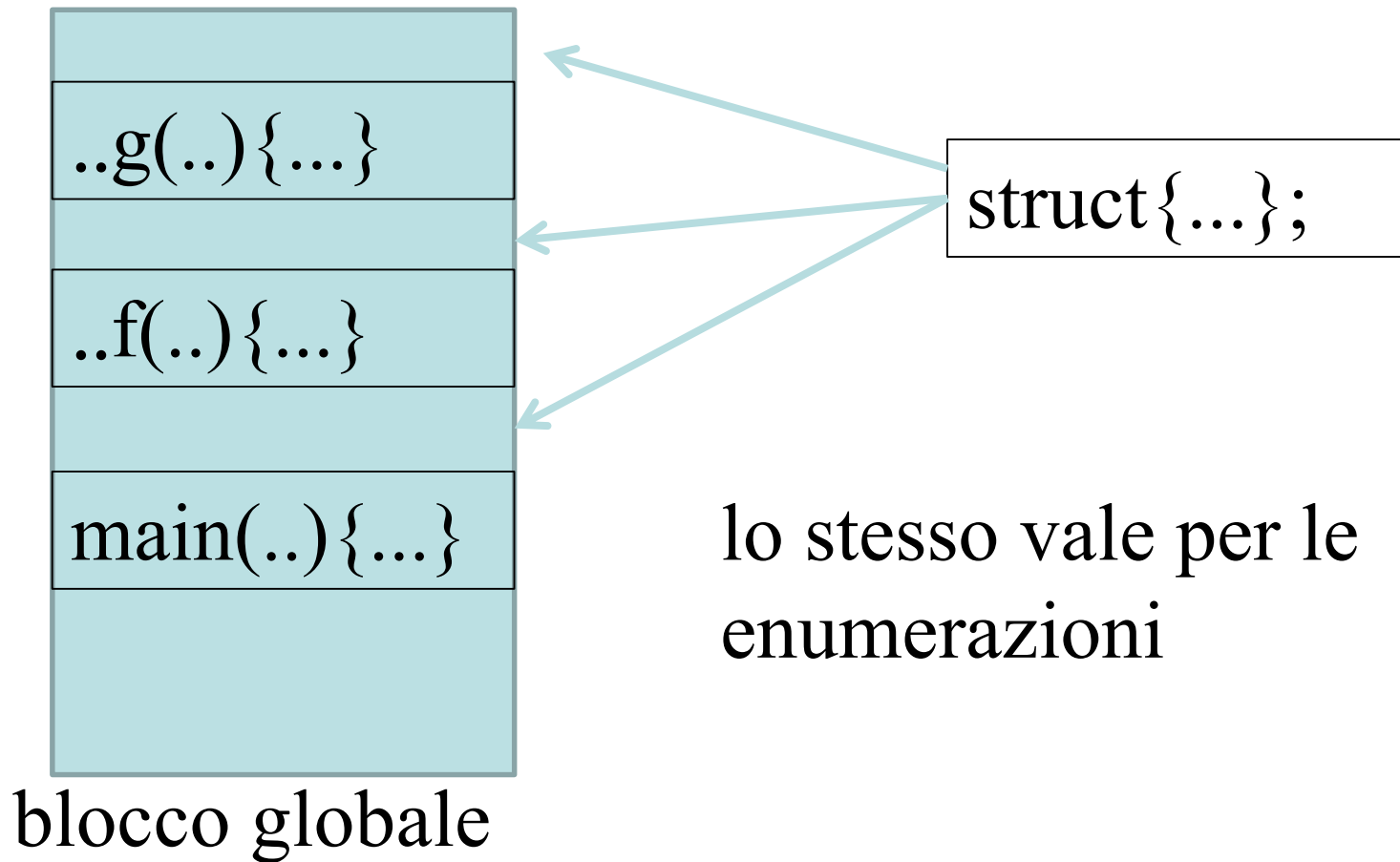
```
x.giorno=27;
```

```
x.mese=3;
```

```
x.anno=2017;
```

operatore • (punto) seleziona un campo

dove vanno posizionate le dichiarazioni di tipo



se i tipi sono definiti in un blocco interno, per esempio in una funzione, allora sono visibili solo in quel blocco

dichiarazioni di strutture ... che succede veramente?

data y; // che valore hanno i campi ?

viene chiamata una funzione, detta costruttore,
che alloca lo spazio di memoria per 3 interi (12
byte) e lascia i campi indefiniti

si tratta del costruttore di default

in questo modo il C++ integra il nuovo tipo nel
linguaggio

c'è anche un altro costruttore offerto dal C++:

il costruttore di copia

supponiamo di avere:

data y;

y.giorno=31; y.mese=3, y.anno=2020;

data z(y);

crea z e copia i campi di y nei corrispondenti campi di z

costruttore senza parametri (di default)

`data()`

e costruttore di copia

`data(const data &)`

sono definiti automaticamente dal C++ per ogni nuova struttura che viene introdotta

il C++ ci permette di scrivere noi i costruttori che ci piacciono

per esempio per la struttura data

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

```
data(int a, int b, int c) {giorno=a; mese=b; anno=c;}  
};
```

esempio d'uso: `data x(1,4,2020);`

ma attenzione !

`data z;` // ora da errore ????

cosa è successo?

il costruttore di default `data()` non esiste più dato che abbiamo introdotto un costruttore nostro

invece il costruttore di copia

`data(const data &)` esiste ancora

ma possiamo ridefinire anche quello, se non ci andasse bene.

semplice trucco per evitare il problema:

```
data(int a=0, int b=0, int c=0)
```

quindi abbiamo contemporaneamente il costruttore con 0, con 1, con 2 e con 3 parametri

valori di default per i parametri formali => 7.2
del testo

la «generosità» del C++ per i tipi struttura continua anche per altre operazioni:

per l'assegnazione

data x(1,4,2020), y; y=x; funziona

cosa fa l'assegnazione di default ?

y=x;

y.giorno= x.giorno;

y.mese=x.mese;

y.anno=x.anno;

cioè copia campo per campo

ci va sempre bene ? Spesso si, ma a volte no,
per esempio con i puntatori ci possono essere
problemi:

```
struct EX {int a, *b; EX(int x) {a=x; b=&a;} };
```

```
EX w(1),q(2);
```

```
w=q; // che succede ?
```


probabilmente preferiamo

```
EX & EX::operator=(const EX & x)
```

```
{a=x.a; b=&a;}
```

che evita di copiare anche il puntatore

per integrare il nuovo tipo possiamo definire

confronto: `EX::operator<(const EX &)`

e la stampa:

`operator<<(const EX &)`

vediamo un esempio “serio” di struttura e la sua integrazione nel linguaggio

ma prima abbiamo bisogno dello switch:

```
switch(espressione){  
case v1: .....;break;  
case v2: .....; break;  
.....  
default: .....;  
}
```

espressione deve avere valore discreto:
enumerazione, char, int,
non float o double e neppure struct

vogliamo rappresentare figure
geometriche colorate

```
enum colore {rosso, bianco, giallo, verde, blu};
```

```
enum figura {triangolo, quadrato, rombo};
```

```
struct punto {int x, y;
```

```
punto(int a=0, int b=0){x=a; y=b;} };
```

```
punto q(1,2);
```

una forma è una figura con un colore e con le
posizioni dei vertici:

```
struct form {figura Fig; colore Col; int nV;  
punto POS[4]};
```

```
form K;
```

```
K.Fig=quadrato;
```

```
K.Col=giallo;
```

```
K.nV=4;
```

```
K.POS[0]=punto(1,1);.....
```

input/output ???

```
cout<< K ;    //non VA
```

dobbiamo farlo noi:

```
cout<< K.Fig<<' '<<<K.Col<' '<<< ..
```

e otteniamo solo interi !!

ma in realtà possiamo ridefinire << per il nostro tipo form

<< per form

```
ostream & operator <<(ostream & s, form & E)
{
    s << "Forma ";
    switch(E.Fig) {
        case triangolo: s << "triangolo" << endl; break;
        case quadrato: s << "quadrato" << endl; break;
        case rombo: s << "rombo" << endl; break;
        default: s << "we got a problem" << endl;
    }
    switch (E.Col) {
        case rosso : s << "rosso" << endl; break;
        case bianco: s << "bianco" << endl; break; // ecc.
```

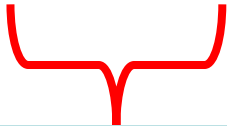
```
for(int i=0; i< E.nV;i++)  
    s << E.POS[i]<<endl // << di punto  
return s;  
}
```

perché ritorna l'ostream s ?

vediamo << che usiamo sempre:

```
cout << x << y ;
```

```
(cout << x) << y;
```



restituisce cout : `cout << y`

```
struct form {figura Fig; colore Col; int nV;  
punto POS[4]};
```

```
form K, *P=&K; //P punta a struttura form
```

```
(*P).Fig=triangolo;
```

oppure con ->

```
P->Fig =triangolo;
```

```
P->Col=giallo;
```

Altro esempio d'uso delle strutture:

visto che passare alle funzioni array con 2 o più dimensioni è poco elastico, possiamo «incartare» gli array in una struttura

A[5][100] e B[4][1000] li possiamo incartare in questa struttura

```
struct A2 {int *a, righe, colonne;} x,y;
```

così:

```
x.a=*A; x.righe=5; x.colonne=100;
```

```
y.a=*B; y.righe=4; y.colonne=1000;
```

se dotiamo A2 di un costruttore:

```
struct A2 {int *a, righe, colonne;  
A2 (int* x=0, int y=0, int z=0) {a=x; righe=y;  
    colonne=z;}  
};
```

```
int A[5][100]      B[4][1000];
```



```
A2 x(*A,5,100), y(*B,4,1000), z;
```

```
int & give(A2 x, int i, int j)
{
    if(i>=x.righe || j>=x.colonne)
        throw(...) // eccezione
    return *(x.a+(i*x.colonne)+j);
}
```

```
A2 z(*B, 5, 1000);
int & ele = give(z,4,3);
```

overloading o sovraccaricamento (9.8)

```
void print(int);
```

```
void print(const char*);
```

```
void print(double);
```

```
void print(long int);
```

```
void print(char);
```

```
char c; float f; short int i;
```

```
print(c); print(f); print(i); print("a");
```

le conversioni hanno un costo:

1) perfetta uguaglianza

2) promozioni

3) contrario delle promozioni

4) conversione definita dall'utente

una invocazione $f(e1, e2)$ e 2 funzioni:

$$f(T1\ x, T2\ y) \leftarrow [c1, c2] \leftarrow (e1, e2)$$
$$f(H1\ z, H2\ w) \leftarrow [h1, h2] \leftarrow (e1, e2)$$

per scegliere la f da invocare, confrontiamo $[c1, c2]$ e $[h1, h2]$

$[c1, c2]$ è meglio di $[h1, h2]$ se

- $c1 \leq h1$ e $c2 \leq h2$

- e inoltre o $c1 < h1$ o $c2 < h2$ o entrambe

se né $[c1, c2]$ è meglio di $[h1, h2]$, né viceversa

allora ambiguità \rightarrow ERRORE del compilatore

algoritmo di overloading resolution

viene applicato dal compilatore

per scegliere la «migliore» funzione per ogni
invocazione

per esempio per operator<<