

tipi definiti dall'utente

testo Cap. 9.2

1) tipo enumerazione

2) tipi strutture

i giorni della settimana:

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

```
giorni x;
```

```
x= martedì;
```

```
.....
```

```
lunedì =0, martedì =1, mercoledì=2,...
```

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

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

conversione 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

sia chiaro:

i tipi enumerazione consentono "solo" di introdurre nel programma delle costanti mnemoniche


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 (sono sostituite con i corrispondenti interi)

tipi strutture

porta d'accesso agli oggetti del C++

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

 **campi** della struttura

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

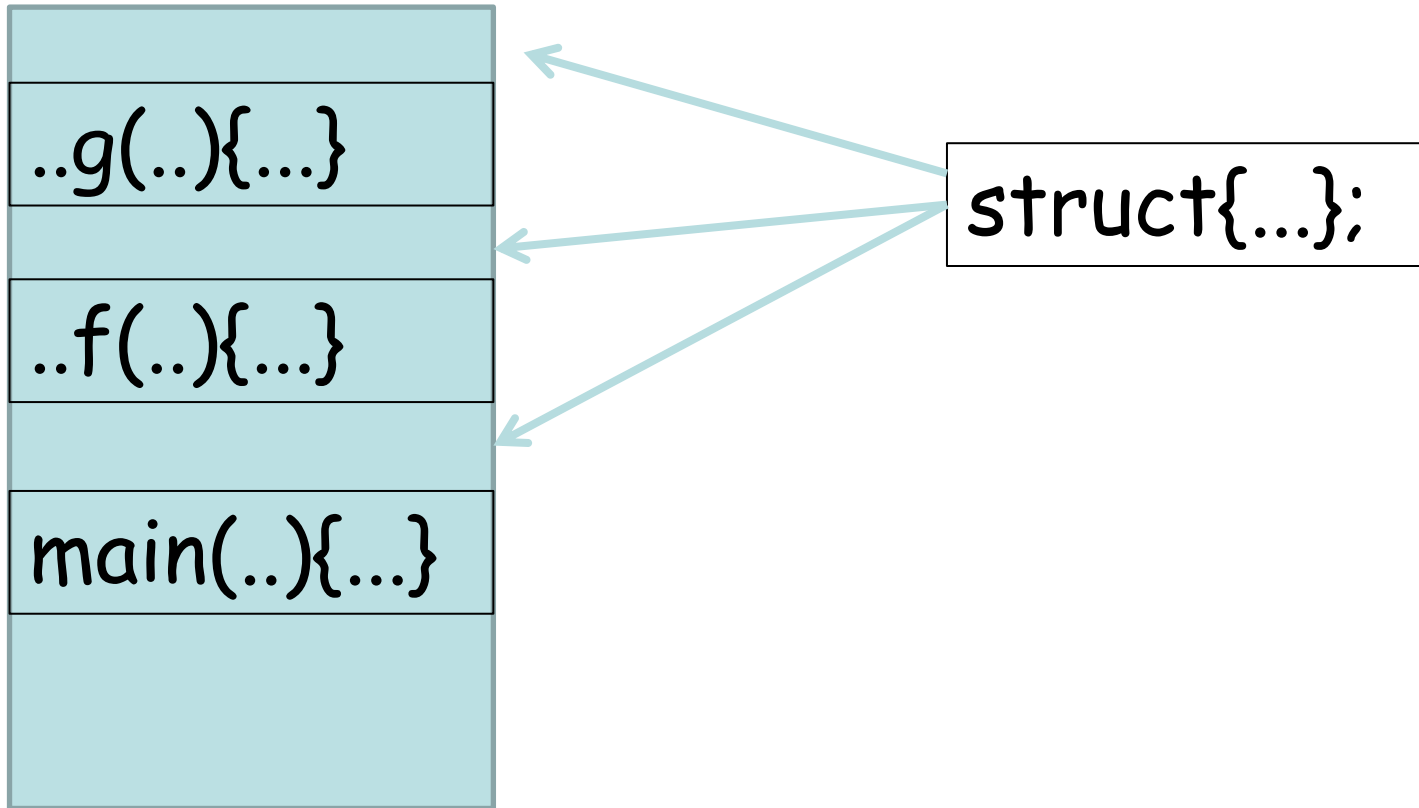
```
x.giorno=8;
```

```
x.mese=2;
```

```
x.anno=2005;
```

operatore . (punto) **seleziona** un campo

dove va scritta una dichiarazione: nel blocco globale (testo 9.7)



dichiarazioni... che succede ?

`data y;` // che valore hanno i campi ?

viene chiamato un **costruttore** che lascia i campi indefiniti `data()`

`y.giorno=12; y.mese=2, y.anno=2008;`

`data z(y);`

si invoca un **costruttore** che copia i campi di y nei corrispondenti campi di z

`data(const data &)`

costruttore senza parametri (di default)

`data()`

e costruttore di copia

`data(const data &)`

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

ma possiamo scrivere noi dei costruttori più interessanti

per esempio per la struttura data

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

e lo usiamo con

```
data x(1,2,2008);
```

ma attenzione !

data z; non va, il costruttore data() non esiste più

invece data(const data &) esiste ancora

semplice trucco per evitare il problema:

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

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

valori di default per i parametri formali => testo 7.2

la *regalia* del C++ per i tipi struttura continua per certe operazioni:

per l'assegnazione

data x(2,3,2008), 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::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

in modo simile possiamo definire anche

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

e la stampa:

```
ostream & operator<<(ostream &, const  
EX &)
```

dichiarazioni di tipo struttura e di
variabili di tipo struttura:

```
struct S{....};
```

```
S x, y; // x e y sono di tipo S
```

struct possono essere innestati:

```
struct P{ .....
```

```
S k; // k ha tipo struct S
```

```
};
```


secondo esempio, enum+struct assieme:

vogliamo rappresentare figure
geometriche colorate

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

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

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

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

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

```
struct form {figura F; colore C; int n_v;  
punto POS[4]};
```

```
form K;
```

```
K.F=quadrato;
```

```
K.C=giallo;
```

```
K.n_v=4;
```

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

input/output ???

`cout<< K ;` ?????? dobbiamo farlo noi
campo per campo

`cout<< K.F<<' '<<K.C<' '<< ..`

e otteniamo solo interi !!

ma in realtà possiamo ridefinire << per
il nostro tipo form
cioè interveniamo nell'overloading

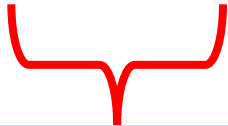
sovraccaricamento di <<

```
ostream & operator<<(ostream & s, form & E)
{
    s << "Forma ";
    switch(E.F)
    {case quadrato: s << "quadrato"<<endl; break;
     case triangolo: s << "triangolo"<<endl; break;
     ....};
    switch(E.C)
    {case rosso:.....}
    for(int i=0; i< E.n_v;i++) s << E.POS[i].x;
    return s;
}
```

vediamo << a cui siamo abituati:

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

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



restituisce cout : cout << y

```
struct form {figura F; colore C; int n_v;  
punto POS[4]};
```

notazione:

form K, *P=&K;

(*P).F=triangolo;

oppure:

$P \rightarrow C = \text{giallo};$

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 posso incartare
in una struttura

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

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

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



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

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

Bubble-sort

```
R=(A[0..k-1] è permutazione di vA[0..k-1] )  
&& (A[i+1..k-1] è ordinato) &&  
(A[i+1..k-1] >= A[0..i])  
for(int i=k-1; i>=0; i--)
```

```
F(A,i);
```

$PRE = (\text{definito } vA[0..i])$

$F(A,i)$

$POST = (A[0..i] \text{ è permutazione di } vA[0..i] \text{ e } A[i] \geq A[0..i-1])$