tipi definiti dall'utente

testo Cap. 9.2

1) tipo enumerazione

2) tipo struttura

i giorni della settimana:

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

giorno 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

deve essere chiaro:

i tipi enumerazione consentono “solo” di introdurre nel programma delle costanti mnemoniche (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 (sono sostituite con i corrispondenti interi)

tipi strutture

porta d’accesso agli oggetti del C++

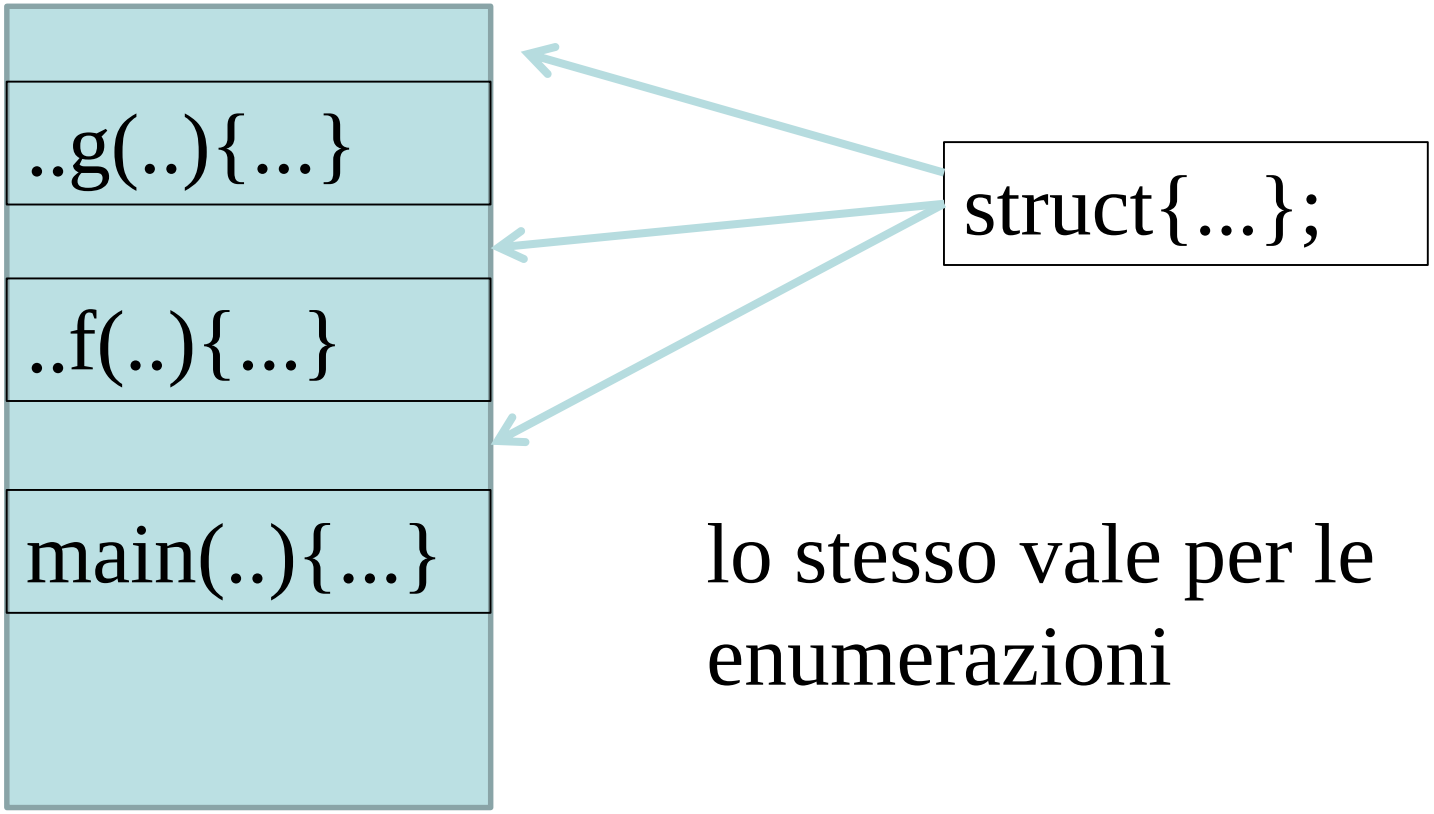
```
struct data {int giorno, mese, anno;};
```

campi della struttura

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

x.giorno=27;

x.mese=3;

x.anno=2017;
```

operatore • (punto) seleziona un campo

dove vanno le dichiarazioni di tipo



lo stesso vale per le enumerazioni

se i tipi sono definiti in un blocco, allora sono visibili sono in quel blocco

dichiarazioni… che succede veramente?

data y; // che valore hanno i campi ?

viene chiamato un costruttore che alloca lo spazio di memoria per 3 interi (12 byte) e lascia i campi indefiniti

si tratta del costruttore di default

c’è anche un altro costruttore: il costruttore di copia

supponiamo di avere:

data y;

y.giorno=1; y.mese=4, y.anno=2017;

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 struttura che viene introdotta

ma possiamo scrivere noi dei costruttori più interessanti

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(1,4,2017);

ma attenzione !

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

cosa è successo?

il costruttore data() non esiste più ora che abbiamo introdotto un costruttore nostro

invece il costruttore di copia

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 anche per altre operazioni:

per l'assegnazione

data x(1,4,2017), 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 & operator=(const EX & x)

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

che evita di copiare anche il puntatore

in modo simile possiamo definire anche

bool operator<(const EX &)

e la stampa:

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

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

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 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 ";
       if(E.f==quadrato)
            s <<“quadrato”<<endl;
        else
         if(E.f==triangolo)
            s <<“triangolo”<<endl;
          else
            if(E.f==.....) .....
       if  (E.C==rosso)
       ..............................
```

```
switch(exp)
case v1:  .....;break;
case v2: .....; break;
.....
default: ........
```

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

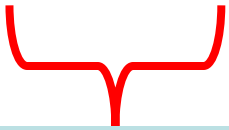
sovraccaricamento di <<

```
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→F =triangolo;

P→C=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 (wrap)

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);
```