47. Le strutture

Una struttura è una collezione ordinata di elementi non omogenei, detti membri o campi. Ciascun campo ha uno specifico tipo, nome e valore.
Le strutture permettono di definire nuovi tipi di oggetti aggregati e oggetti complessi in C++; la struttura può essere utilizzata come un oggetto unico e i campi possono essere utilizzati singolarmente.

Ciascun campo può essere a sua volta un tipo struttura.

47.1 Definizione di un tipo struct

Viene definito un nuovo tipo aggregato.
La definizione viene fatta **globalmente!** (infatti sostanzialmente definiamo un nuovo tipo di variabile costituito da più tipi).

Sintassi

```
struct new_struct_id {        //si definisce, globalmente, il nuovo "tipo" new_struct_id"
tipo1 campo1;
...
tipoN campoN;
};

new_struct_id var_id;         //si crea una nuova variabile var_id non di un tipo standard, ma del
                              tipo new_struct_id sopra definito. La variabile dipenderà dagli N
                              campi.
```

Esempio

```
struct complex {              // definizione del tipo "complex"
double re;                    // campo "reale"
double im;                    // campo "immaginario"
};

complex c,c1;                 // definizione di variabili di tipo "complex"
```

47.2 Strutture annidate

Come detto in precedenza, i campi delle strutture possono essere a loro volta dei campi.

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

struct persona {                    // struttura annidata
char nome[25], cognome[25];
char comune_nascita[25];
data data_nascita;                  //data_nascita è una variabile del tipo data
enum { F, M } sesso;
};

struct studente {                   // struttura ulteriormente annidata
persona generalita;                 //generalita è una variabile del tipo persona
char matricola[10];
int anno_iscrizione;
};
```

Esempio di accesso (due strutture: *struct dati{char nome[10], cognome[10];*); e *struct studente {float media;dati generalita};*
dati informazioni={"Piero","Fattore"}; studente s;
s.media=21;
strcpy(s.generalita.nome,informazioni.nome);
cin >> s.generalita.cognome;

cout << s.generalita.nome << s.generalita.cognome;
//cout << s.generalita; ***errore***

47.3 Inizializzazioni di variabili di tipo *struct*
Una variabile di tipo *struct* viene inizializzata con liste ordinate dei valori dei campi rispettivi. Devono combaciare per ordine e tipo. Eventuali valori mancanti vengono iniziati allo zero del tipo.

Sintassi
new_struct_id var_id = {exp,exp1,exp2,exp3,exp4};

//le varie *exp* devono essere dei valori conformi al tipo di campo cui corrispondono (se sono una stringa saranno *"Ciao"*, un carattere *'c'*, un numero *9*, …)

Esempi
Definiamo una variabile *x* del tipo *persona*

persona x = {"Paolo", "Rossi","Trento",{21,10,1980}, M};

//osserviamo che il terzo elemento è una struttura e quindi scriviamo una lista di elementi, che corrispondono per ordine e tipo al tipo della struttura cui ci riferiamo

Al posto delle costanti si possono anche mettere delle variabili:
int a=21, b=10, c=1980;
data_nascita date = {a,b,c};

Se dobbiamo assegnare una stringa ad un campo di una struttura con una variabile, non mettiamo il nome della stringa nella lista ma facciamo così.

char nome[]="Paolo";
persona x = {"AA","Rossi","Trento",{21,10,1980},M};
//oppure
strcpy(x.nome,nome);

È comunque consigliato inizializzare una stringa con una lista solo quando si hanno costanti, in tutti gli altri casi conviene operare direttamente sul campo, come fosse una comune variabile di tipo noto.

! Si possono anche definire array di una struttura ! Infatti al pari di una variabile posso fare:
persona insieme_di_persone[10] = {};
insieme_di_persone[0].nome; //il nome della prima persona
insieme_di_persone[5].conome; //il cognome della sesta persona

47.4 Accesso ai campi di una struct.
L'assegnazione ai campi di una struct, non può essere fatta con un ciclo *for*, quindi bisogna fare *n* volte le operazioni di accesso ai campi di una struct e modifica, dove *n* è il numero dei campi.

- Se *s* è una struct e *field* è un identificatore di un suo campo, allora *s.field* denota il campo della struct: ***s.field*** è un'espressione dotata di indirizzo! Può essere letta con >>, assegnata, passata per riferimento, ecc.
- Se *ps* è un puntatore ad una struttura avente *field* come campo, possiamo far puntare il nostro puntatore a *s* (facendo la solita assegnazione dei puntatori ***new_struct_id* ps = &s***) e poi manipolare *field* in questo modo: ***ps->field*** (zucchero sintattico molto usato).
  ps->field può essere usato analogamente a *s.field*.

Esempio
struct complex { double re, im; };
*complex c; complex *pc = &c;*

c.re = 2.5; pc->im = 3;
cin >> c.re >> c.im;
cout << c.re << endl << c.im,
swap(c.re,c.im);
c.re = a;
int array[10];
cin >> array[1];
c.re=array[1];

47.5 Assegnazione tra strutture
A differenza degli array, l'assegnazione tra struct è definita.

L'assegnazione di struct avviene per valore. Vengono copiati tutti i valori dei membri.
Se un campo è un array statico, viene copiato per intero!

La copia di struct può essere computazionalmente onerosa!

persona x,y = {"Paolo", "Rossi", "Trento", {21,10,1980}, M };
x=y; // vengono copiate tutte le stringhe!

47.6 Passaggio di strutture a funzioni
A differenza degli array, le strutture:
- possono essere passate per valore ad una funzione
- possono essere restituire da una funzione tramite *return*

Entrambe le operazioni comportano una copia dei valori dei membri (array compresi!). Entrambe le operazioni possono essere computazionalmente onerose!

Quando possibile, è preferibile utilizzare **passaggio per riferimento** (con *const*).

Esempio
void stampa_persona (persona p) {...}
void stampa_persona1 (const persona & p) {...}
persona x,y = {"Paolo", "Rossi", "Trento", {21,10,1980}, M };

stampa_persona(y); // viene fatta una copia
stampa_persona1(y); // non viene fatta alcuna copia

Esempio di passaggio e ritorno di una struttura (*struct punto {float x, y;};*)
punto trasladiuno(const punto&);

Nel *main*: *punto p, q;* - inizializza *p* - *q=trasladiuno(p);* - stampa *q*.
La funzione:

```
punto trasladiuno(const punto& puntoIN){
  punto puntoFIN;
  puntoFIN.x=puntoIN.x+1;
  puntoFIN.y=puntoIN.y+1;
return puntoFIN;}
```

47.7 Assegnazione di array statici tramite struct
Per gestire gli array statici in modo che possano essere copiati, si può "incapsulare" un tipo array come membro di una *struct*.
Esempio

```
struct int_array { int ia[3]; };          //struttura con un solo elemento per gestire gli array
int_array sa, sb;

sa.ia[0]=1;
sa.ia[1]=2;
sa.ia[2]=3;

sb = sa; // l'array viene copiato!

int_array sc[3]; //un array di dimensione 3 che contiene 3 array

cin >> sc[0].ia[0] >> sc[0].ia[1] >> sc[0].ia[2];
```

47.8 Assegnazione di array dinamici tramite struct
Nel caso di array dinamici, viene copiato solo il puntatore. Nella definizione della struttura si mette come campo un *tipo** (un puntatore, a cui verrà poi assegnata un'area di memoria della heap).

```
struct int_array { int * ia; };
sa.ia = new int[3];
sb = sa;
```

viene copiato il puntatore, ***sa.ia e sb.ia sono lo stesso array!*** Per questo basta fare un solo *delete* alla fine del programma.

47.9 Strutture ricorsive
La seguente definizione non è lecita:

```
struct S {
int value;
S n; //definizione circolare!
};
```

La seguente definizione è lecita:

```
struct S {
int value;
S *n;
};
```

Ogni puntatore occupa lo stesso spazio di memoria indipendentemente dal suo tipo. Molto importante per strutture dati dinamiche!

Strutture mutualmente ricorsive

La seguente definizione non è lecita:

```
struct S1
{ int value;
S2 *n; };  //S2 ancora indefinito

struct S2
{ int value;
S1 *n; };
```

La seguente definizione è lecita:

```
struct S2; //dichiarazione di S2

struct S1
{ int value;
S2 *n; };        // Ok!

struct S2        // definizione di S2
{ int value;
S1 *n;};
```

48 Uso di array ordinati di strutture
È frequente il dover gestire archivi ordinati di oggetti complessi (ES: persone, articoli, libri, ecc.).

L'uso di strutture rappresenta l'elemento base dei sistemi informativi (es: archivi, inventari, rubriche, anagrafi, ecc.). Per l'ordinamento usa qualche campo specifico (chiave).

Tipicamente utilizzate strutture di dati dinamiche ad hoc (es: alberi di ricerca binaria).

Esempio di archivio semplificato: array ordinato di persone.

Esempio 1:
Array ordinato di strutture *persona*:
persona persone [NmaxPers];

Ordinamento e ricerca usano strcmp sul campo cognome:
if (strcmp(p[i].cognome,cognome) < 0) → ricerca binaria
if(strcmp(p[i].cognome,p[i+1].cognome) <0) → bubble sort

Problema: ogni swap effettua 3 copie tra struct! Molto inefficiente!

Esempio 2:
Array ordinato di **puntatori** a strutture *persona:*
persona * persone [NmaxPers];

Ordinamento e ricerca usano strcmp sul campo cognome:
if (strcmp(p[i]->cognome,cognome) < 0) → ricerca binaria
if(strcmp(p[i]->cognome,persona[i+1]->cognome) <0) → bubble sort

Importante: ogni swap effettua 3 copie tra puntatori. Efficiente! Ogni puntatore ha dimensione fissa, indipendentemente dal tipo puntato.

Esempio 3:
Doppio array ordinato di puntatori a persone
Richiesta: poter effettuare ricerca sia per nome che per cognome.

Idea: 2 array di puntatori a strutture "persona", ordinati rispettivamente per nome e cognome.

*persona * nomi [NmaxPers];*
*persona * cognomi [NmaxPers];*

Ordinamento e ricerca usano strcmp sul campo cognome e nome rispettivamente:
if (strcmp(p[i]->cognome,cognome) < 0) ...
if (strcmp(p[i]->nome,nome) < 0) …

Le struct sono condivise tra i due array, ogni swap effettua 3 copie tra puntatori. Efficiente!

Pattern per gestire tante strutture

```
struct persone {char nome[20], cognome[20], sesso[2];};

int main () {

  int totpersone=4;
  persone persona[4];

  for(int i=0;i<4;i++){
    cout << "Dati persona " << i << endl;
    cin >> persona[i].nome >> persona[i].cognome >> persona[i].sesso;
    cout << endl;
  }

  for(int i=0;i<4;i++){
    cout << "Dati persona " << i << endl;
    cout << persona[i].nome << endl;
    cout << persona[i].cognome << endl;
    cout << persona[i].sesso << endl;
    cout << endl;
  }

}
```

Può essere implementato con un menu' testuale e delle funzioni e l'utente può arbitrariamente:

- decidere di stampare la persona *i* o una persona precisa cercata per cognome
- inserire un'altra persona (notificare l'utente se il database è pieno).
- eliminare una persona (cerca la persona per cognome e elimina la cella *i* dell'array persona poi le altre celle scalano)
- stamparle tutte in ordine alfabetico (bubble sort, prediligendo il cognome).

! Passare le strutture per riferimento ! Oppure passo un puntatore se ho un array dinamico.

Si può anche usare al posto di *cin* e *cout* un file di testo ben formattato.