

Corso “Programmazione 1”

Capitolo 09: Le Strutture

Docente: **Roberto Sebastiani** - roberto.sebastiani@unitn.it
Esercitori: **Mario Passamani** - mario.passamani@unitn.it
Alessandro Tomasi - alessandro.tomasi@unitn.it
C.D.L.: Informatica (INF)
Ing. Informatica, delle Comunicazioni ed Elettronica (ICE)
Studenti con numero di matricola pari
A.A.: 2019-2020
Luogo: DISI, Università di Trento
URL: disi.unitn.it/rseba/DIDATTICA/prog1_2020/

Outline

1 Le Strutture

2 Operazioni su Strutture

3 Strutture Ricorsive

4 Array ordinati di Strutture

1 Le Strutture

2 Operazioni su Strutture

3 Strutture Ricorsive

4 Array ordinati di Strutture

Le Strutture

- Una struttura è una **collezione ordinata di elementi non omogenei**
 - gli elementi sono detti **membri** o **campi**
 - ciascun campo ha uno specifico **tipo**, **nome** e **valore**
- Permette di definire **nuovi tipi di oggetti aggregati**
 - La struttura può essere utilizzata come un oggetto unico
 - I campi possono essere utilizzati singolarmente
- Ciascun campo può essere a sua volta un tipo struct

Definizione di un tipo struct

- Viene definito un nuovo **tipo** aggregato

- Sintassi:

```
struct new_struct_id {  
    tipo1 campo1;  
    ...  
    tipoN campoN;  
};  
new_struct_id var_id;
```

- Esempio:

```
struct complex { // definizione del tipo "complex"  
    double re; // campo "reale"  
    double im; // campo "immaginario"  
};  
complex c, c1; // definizione di variabili  
                // di tipo "complex"
```

Alcuni esempi di strutture annidate

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

```
struct persona { // struttura annidata  
    char nome[25], cognome[25];  
    char comune_nascita[25];  
    data data_nascita;  
    enum { F, M } sesso;  
};
```

```
struct studente { // struttura ulteriormente annidata  
    persona generalita;  
    char matricola[10];  
    int anno_iscrizione;  
};
```

Inizializzazione 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

```
struct data {
    int giorno, mese, anno;
};
struct persona {
    char nome[25], cognome[25];
    char comune_nascita[25];
    data data_nascita;
    enum { F, M } sesso;
};
persona x = {"Paolo", "Rossi", "Trento",
            {21,10,1980}, M };
```

Accesso ai campi di una struttura

- Se `s` è una struttura e `field` è un identificatore di un campo, allora `s.field` denota il campo della struttura
 - `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, allora è possibile scrivere `ps->field` al posto di `(*ps).field`
 - zucchero sintattico
 - uso molto frequente

Esempio

```
struct complex { double re, im; };
complex c; complex *pc = &c;
c.re = 2.5; pc->im = 3;
cin >> c.re >> c.im;
swap(c.re, c.im);
```


Accesso ai campi di una struttura

- Se `s` è una struttura e `field` è un identificatore di un campo, allora `s.field` denota il campo della struttura
 - `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, allora è possibile scrivere `ps->field` al posto di `(*ps).field`
 - zucchero sintattico
 - uso molto frequente

Esempio

```
struct complex { double re, im; };
complex c; complex *pc = &c;
c.re = 2.5; pc->im = 3;
cin >> c.re >> c.im;
swap(c.re, c.im);
```

Accesso ai campi di una struttura

- Se `s` è una struttura e `field` è un identificatore di un campo, allora `s.field` denota il campo della struttura
 - `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, allora è possibile scrivere `ps->field` al posto di `(*ps).field`
 - zucchero sintattico
 - uso molto frequente

Esempio

```
struct complex { double re, im; };
complex c; complex *pc = &c;
c.re = 2.5; pc->im = 3;
cin >> c.re >> c.im;
swap(c.re, c.im);
```

Esempi

- operazioni di base sui membri di struct :
{ STRUCT/struct.cc }
- ... con inizializzazione:
{ STRUCT/struct1.cc }

Esempi

- operazioni di base sui membri di struct :
{ STRUCT/struct.cc }
- ... con inizializzazione:
{ STRUCT/struct1.cc }

Outline

1 Le Strutture

2 Operazioni su Strutture

3 Strutture Ricorsive

4 Array ordinati di Strutture

Assegnazione di Strutture

- A differenza degli array, l'**assegnazione tra struct** è definita
- L'assegnazione di strutture 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!
```

Assegnazione di Strutture

- A differenza degli array, l'**assegnazione tra struct** è definita
- L'assegnazione di strutture 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!
```

Assegnazione di Strutture

- A differenza degli array, l'**assegnazione tra struct** è definita
- L'assegnazione di strutture 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!
```


Assegnazione di Strutture

- A differenza degli array, l'**assegnazione tra struct** è definita
- L'assegnazione di strutture 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!
```

Passaggio di Strutture a Funzioni

- A differenza degli array, le strutture:
 - possono essere passate per valore ad una funzione
 - possono essere restituite da una funzione tramite `return`
- entrambe le operazioni comportano una **copia** dei valori dei membri (array compresi!)

⇒ **entrambe possono essere computazionalmente onerose!**

⇒ Quando possibile, è preferibile utilizzare passaggio per riferimento (con `const`)

```
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
```

Passaggio di Strutture a Funzioni

- A differenza degli array, le strutture:
 - possono essere passate per valore ad una funzione
 - possono essere restituite da una funzione tramite `return`
- entrambe le operazioni comportano una **copia** dei valori dei membri (array compresi!)

⇒ entrambe possono essere computazionalmente onerose!

⇒ Quando possibile, è preferibile utilizzare passaggio per riferimento (con `const`)

```
void stampa_persona (persona p) {...}
void stampa_personal (const persona & p) {...}

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

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

Passaggio di Strutture a Funzioni

- A differenza degli array, le strutture:
 - possono essere passate per valore ad una funzione
 - possono essere restituite da una funzione tramite `return`
- entrambe le operazioni comportano una **copia** dei valori dei membri (array compresi!)

⇒ **entrambe possono essere computazionalmente onerose!**

⇒ Quando possibile, è preferibile utilizzare passaggio per riferimento (con `const`)

```
void stampa_persona (persona p) {...}
void stampa_personal (const persona & p) {...}

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

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

Passaggio di Strutture a Funzioni

- A differenza degli array, le strutture:
 - possono essere passate per valore ad una funzione
 - possono essere restituite da una funzione tramite `return`
- entrambe le operazioni comportano una **copia** dei valori dei membri (array compresi!)

⇒ **entrambe possono essere computazionalmente onerose!**

⇒ Quando possibile, è preferibile utilizzare passaggio per riferimento (con `const`)

```
void stampa_persona (persona p) {...}
void stampa_personal (const persona & p) {...}

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

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

Passaggio di Strutture a Funzioni

- A differenza degli array, le strutture:
 - possono essere passate per valore ad una funzione
 - possono essere restituite da una funzione tramite `return`
- entrambe le operazioni comportano una **copia** dei valori dei membri (array compresi!)

⇒ **entrambe possono essere computazionalmente onerose!**

⇒ Quando possibile, è preferibile utilizzare passaggio per riferimento (con `const`)

```
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
```

Esempi

- assegnazione e passaggio di struct :
{ STRUCT/struct3.cc }

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

```
struct int_array { int ia[3]; };  
int_array sa, sb;  
sa.ia[0]=1;sa.ia[1]=2;sa.ia[2]=3;  
sb = sa; //l'array viene copiato!
```

- ... con inizializzazione:

```
{ STRUCT/struct_array1.cc }
```


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

```
struct int_array { int ia[3]; };  
int_array sa, sb;  
sa.ia[0]=1;sa.ia[1]=2;sa.ia[2]=3;  
sb = sa; //l'array viene copiato!
```

- ... con inizializzazione:
{ STRUCT/struct_array1.cc }

Assegnazione di array dinamici tramite struct

Nel caso di **array dinamici**, viene copiato solo il puntatore

```
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!
```

- esempio di cui sopra, esteso:
{ STRUCT/struct_arraypunt.cc }

Assegnazione di array dinamici tramite struct

Nel caso di **array dinamici**, viene copiato solo il puntatore

```
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!
```

- esempio di cui sopra, esteso:
{ STRUCT/struct_arraypunt.cc }

Outline

1 Le Strutture

2 Operazioni su Strutture

3 Strutture Ricorsive

4 Array ordinati di Strutture

Strutture ricorsive

- La seguente definizione non è lecita:

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

- La seguente definizione è lecita:

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

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

Strutture ricorsive

- La seguente definizione non è lecita:

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

- La seguente definizione è lecita:

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

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

Strutture ricorsive

- La seguente definizione non è lecita:

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

- La seguente definizione è lecita:

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

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

Strutture ricorsive

- La seguente definizione non è lecita:

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

- La seguente definizione è lecita:

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

- Infatti 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 *next; }; //S2 ancora indefinito
struct S2
{ int value;
  S1 *next; };
```

- La seguente definizione è lecita:

```
struct S2; // dichiarazione di S2
struct S1
{ int value;
  S2 *next; }; // Ok!
struct S2 // definizione di S2
{ int value;
  S1 *next;};
```

Strutture mutualmente ricorsive

- La seguente definizione non è lecita:

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

- La seguente definizione è lecita:

```
struct S2; // dichiarazione di S2
struct S1
{ int value;
  S2 *next; }; // Ok!
struct S2 // definizione di S2
{ int value;
  S1 *next;};
```

Esempi

- **struttura ricorsiva, non corretta:**
{ STRUCT/rec_struct_err.cc }
- **strutt ricorsiva, corretta:**
{ STRUCT/rec_struct.cc }
- **strutture mutualmente ricorsive, non corrette:**
{ STRUCT/mutrec_struct_err.cc }
- **strutture mutualmente ricorsive, corrette:**
{ STRUCT/mutrec_struct.cc }

Esempi

- **struttura ricorsiva, non corretta:**
{ STRUCT/rec_struct_err.cc }
- **strutt ricorsiva, corretta:**
{ STRUCT/rec_struct.cc }
- **strutture mutualmente ricorsive, non corrette:**
{ STRUCT/mutrec_struct_err.cc }
- **strutture mutualmente ricorsive, corrette:**
{ STRUCT/mutrec_struct.cc }

Esempi

- **struttura ricorsiva, non corretta:**
{ STRUCT/rec_struct_err.cc }
- **strutt ricorsiva, corretta:**
{ STRUCT/rec_struct.cc }
- **strutture mutualmente ricorsive, non corrette:**
{ STRUCT/mutrec_struct_err.cc }
- **strutture mutualmente ricorsive, corrette:**
{ STRUCT/mutrec_struct.cc }

Esempi

- **struttura ricorsiva, non corretta:**
{ STRUCT/rec_struct_err.cc }
- **strutt ricorsiva, corretta:**
{ STRUCT/rec_struct.cc }
- **strutture mutualmente ricorsive, non corrette:**
{ STRUCT/mutrec_struct_err.cc }
- **strutture mutualmente ricorsive, corrette:**
{ STRUCT/mutrec_struct.cc }

Outline

1 Le Strutture

2 Operazioni su Strutture

3 Strutture Ricorsive

4 Array ordinati di Strutture

Uso di Array Ordinati di Strutture

- È frequente il dover gestire **archivi ordinati** di oggetti complessi (ES: persone, articoli, libri, ecc.)
 - Elemento base dei **sistemi informativi**
(ES: archivi, inventari, rubriche, anagrafi, ecc.)
 - 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**

Uso di Array Ordinati di Strutture

- È frequente il dover gestire **archivi ordinati** di oggetti complessi (ES: persone, articoli, libri, ecc.)
 - Elemento base dei **sistemi informativi**
(ES: archivi, inventari, rubriche, anagrafi, ecc.)
 - 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**

Uso di Array Ordinati di Strutture

- È frequente il dover gestire **archivi ordinati** di oggetti complessi (ES: persone, articoli, libri, ecc.)
 - Elemento base dei **sistemi informativi**
(ES: archivi, inventari, rubriche, anagrafi, ecc.)
 - 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**

Uso di Array Ordinati di Strutture

- È frequente il dover gestire **archivi ordinati** di oggetti complessi (ES: persone, articoli, libri, ecc.)
 - Elemento base dei **sistemi informativi**
(ES: archivi, inventari, rubriche, anagrafi, ecc.)
 - 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**

Uso di Array Ordinati di Strutture

- È frequente il dover gestire **archivi ordinati** di oggetti complessi (ES: persone, articoli, libri, ecc.)
 - Elemento base dei **sistemi informativi**
(ES: archivi, inventari, rubriche, anagrafi, ecc.)
 - 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 Persone

- Array ordinato di strutture “persona”:

```
persona persone [NmaxPers];
```

- Ordinamento e ricerca usano `strcmp` sul campo `cognome`:

```
if (strcmp(p[i].cognome, cognome) < 0) ...
```

- array ordinato di struct, bubblesort, ricerca binaria:

```
{ STRUCT/persone.cc }
```

- Problema: ogni swap effettua 3 copie tra struct !

⇒ molto inefficiente!

Esempio 1: Array Ordinato di Persone

- Array ordinato di strutture “persona”:

```
persona persone [NmaxPers];
```

- Ordinamento e ricerca usano `strcmp` sul campo `cognome`:

```
if (strcmp(p[i].cognome, cognome) < 0) ...
```

- array ordinato di struct, bubblesort, ricerca binaria:

```
{ STRUCT/persone.cc }
```

- Problema: ogni swap effettua 3 copie tra struct !

⇒ molto inefficiente!

Esempio 1: Array Ordinato di Persone

- Array ordinato di strutture “persona”:

```
persona persone [NmaxPers];
```

- Ordinamento e ricerca usano `strcmp` sul campo `cognome`:

```
if (strcmp(p[i].cognome, cognome) < 0) ...
```

- array ordinato di struct, bubblesort, ricerca binaria:

```
{ STRUCT/persone.cc }
```

- Problema: ogni swap effettua 3 copie tra struct !

⇒ molto inefficiente!

Esempio 1: Array Ordinato di Persone

- Array ordinato di strutture “persona”:

```
persona persone [NmaxPers];
```

- Ordinamento e ricerca usano `strcmp` sul campo `cognome`:

```
if (strcmp(p[i].cognome, cognome) < 0) ...
```

- array ordinato di struct, bubblesort, ricerca binaria:

```
{ STRUCT/persone.cc }
```

- Problema: ogni swap effettua 3 copie tra struct !

⇒ molto inefficiente!

Esempio 2: Array Ordinato di Puntatori a Persone

- 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) ...
```

- array ordinato di puntatori a struct, bubblesort, ricerca binaria:

```
{ STRUCT/persone2.cc }
```

- Importante: ogni swap effettua 3 copie tra puntatori
⇒ efficiente!

Esempio 2: Array Ordinato di Puntatori a Persone

- 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) ...
```

- array ordinato di puntatori a struct, bubblesort, ricerca binaria:

```
{ STRUCT/persone2.cc }
```

- Importante: ogni swap effettua 3 copie tra puntatori
⇒ efficiente!

Esempio 2: Array Ordinato di Puntatori a Persone

- 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) ...
```

- **array ordinato di puntatori a struct, bubblesort, ricerca binaria:**

```
{ STRUCT/persone2.cc }
```

- Importante: **ogni swap effettua 3 copie tra puntatori**
⇒ efficiente!

Esempio 2: Array Ordinato di Puntatori a Persone

- 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) ...
```

- **array ordinato di puntatori a struct, bubblesort, ricerca binaria:**

```
{ STRUCT/persone2.cc }
```

- **Importante: ogni swap effettua 3 copie tra puntatori**
⇒ efficiente!

Es. 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) ...
```

- array ordinato di puntatori a struct, bubblesort, ricerca binaria:

```
{ STRUCT/persone3.cc }
```

- Importante: Le struct sono condivise tra i due array, ogni swap effettua 3 copie tra puntatori

⇒ efficiente!

Es. 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) ...
```

- array ordinato di puntatori a struct, bubblesort, ricerca binaria:
{ `STRUCT/person3.cc` }
- Importante: Le struct sono condivise tra i due array, ogni swap effettua 3 copie tra puntatori
⇒ efficiente!

Es. 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) ...
```

- array ordinato di puntatori a struct, bubblesort, ricerca binaria:

```
{ STRUCT/persone3.cc }
```

- Importante: Le struct sono condivise tra i due array, ogni swap effettua 3 copie tra puntatori

⇒ efficiente!

Es. 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) ...
```

- array ordinato di puntatori a struct, bubblesort, ricerca binaria:
{ STRUCT/person3.cc }

- Importante: Le struct sono condivise tra i due array, ogni swap effettua 3 copie tra puntatori

⇒ efficiente!

Es. 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) ...
```

- array ordinato di puntatori a struct, bubblesort, ricerca binaria:
{ STRUCT/person3.cc }
- Importante: Le struct sono condivise tra i due array, ogni swap effettua 3 copie tra puntatori
⇒ efficiente!

Esercizi proposti

Vedere file `ESERCIZI_PROPOSTI.txt`