



# ELEMENTI DI INFORMATICA

DOCENTE: FRANCESCO MARRA

INGEGNERIA CHIMICA

INGEGNERIA ELETTRICA

SCIENZE ED INGEGNERIA DEI MATERIALI

INGEGNERIA GESTIONALE DELLA LOGISTICA E DELLA PRODUZIONE

INGEGNERIA NAVALE

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

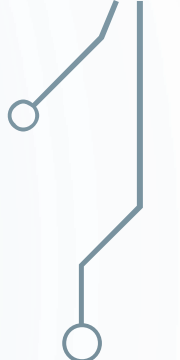
TIPI STRUTTURATI: RECORD

RICERCA BINARIA





# AGENDA

- Tipi strutturati
    - Record
  - Ricerca binaria
- 

# TIPI STRUTTURATI: RECORD



# I RECORD

- Il *record* è una collezione di elementi anche di diverso tipo, a cui viene dato un unico nome
  - Detto anche *struttura*
- Gli elementi sono detti *campi*
  - Possono essere di tipi differenti
- La dichiarazione di un record prevede la specifica di:
  - **Nome** del record
  - **Campi** del record

# DICHIARAZIONE DI UN RECORD

- Dichiarazione di un tipo record:

```
struct Nome_record {  
    tipo_campo1 nome_campo1;  
    ...  
    tipo_campoN nome_campoN;  
};
```

- Oppure mediante typedef

```
typedef struct {  
    tipo_campo1 nome_campo1;  
    ...  
    tipo_campoN nome_campoN;  
} Nome_record;
```

Es.:

```
struct Studente {  
    char nome[20];  
    char cognome[20];  
    int matricola;  
};
```

Es.:

```
typedef struct {  
    char nome[20];  
    char cognome[20];  
    int matricola;  
} Studente;
```

# DICHIARAZIONE DI UN RECORD

- Definizione di una variabile di tipo record

```
struct Nome_record record1, record2;
```

Es.:

```
struct Studente studente1, studente2;
```

- Oppure:

```
Nome_record record1, record2;
```

Es.:

```
Studente studente1, studente2;
```

- È anche possibile usare un'unica dichiarazione di tipo e di variabili

```
struct Nome_record {  
    tipo_campo1 nome_campo1;  
    ...  
    tipo_campoN nome_campoN;  
} record1, record2;
```

Es.:

```
struct Studente {  
    char nome[20];  
    char cognome[20];  
    int matricola;  
} studente1, studente2;
```

# DICHIARAZIONE DI UN RECORD

- È possibile effettuare l'inizializzazione dei campi di un record elencando i valori da assegnare

```
struct Nome_record record = {valore1,...,valoreN};
```

Es.:

```
struct Studente studente1 = {Mario, Bianchi, 450001};
```



# FUNZIONE DI ACCESSO AD UN RECORD

- Una volta definito un record, si può solo operare con i suoi campi accedendo ad essi mediante la **dot notation**

`nome_record.nome_campo`

Es.:

```
studente.nome = ``Mario``;
```

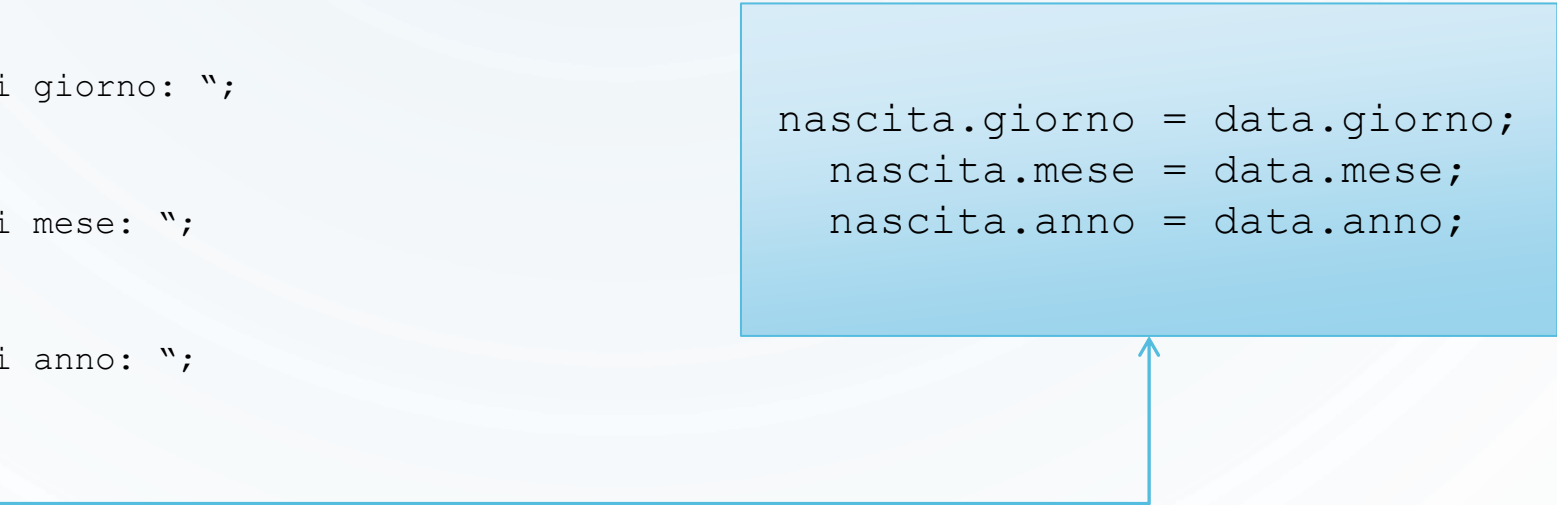
# OPERAZIONI SUI RECORD

- Sui record le uniche operazioni ammesse sono:
  - quelle definite sui singoli campi
  - la copia di un intero record in un altro dello stesso tipo

- **Esempio**

```
cout << "\nInserisci giorno: ";  
cin >> data.giorno;  
cout << "\nInserisci mese: ";  
cin >> data.mese;  
cout << "\nInserisci anno: ";  
cin >> data.anno;  
nascita = data;
```

```
nascita.giorno = data.giorno;  
nascita.mese = data.mese;  
nascita.anno = data.anno;
```



# TIPI DI RECORD

- Strutture definite con definizioni differenti, ma identiche campo per campo, non sono considerate dello stesso tipo
  - La compatibilità è *nominale*
- Strutture con nomi differenti possono contemplare al loro interno campi che hanno gli stessi nomi

```
struct Tipo_data1 {  
    int giorno;  
    char mese[15];  
    int anno;  
} data1;
```

```
struct Tipo_data2 {  
    int giorno;  
    char mese[15];  
    int anno;  
} data2;
```

~~data1 = data2;~~

# STRUTTURE COMPLESSE

```
struct anagrafica {
```

```
    char nome[30];
```

```
    char cognome[40];
```

```
    char codice_fiscale[17];
```

```
    char residenza[100];
```

```
    struct tipo_data {
```

```
        int giorno;
```

```
        char mese[15];
```

```
        int anno;
```

```
    } nascita;
```

```
    int telefono[20];
```

```
};
```

cittadino.nome[0]


cittadino.nascita.giorno

cittadino.telefono[2]

# ARRAY DI RECORD: ESEMPIO

```
struct Calciatore{  
    char nome[30];  
    char cognome[40];  
    char ruolo[20];  
    int eta;  
};  
  
Calciatore Squadra[30];
```

Array di 30  
elementi di  
tipo  
Calciatore




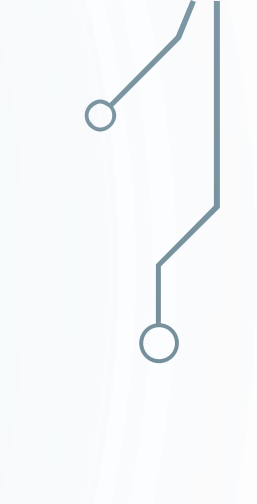
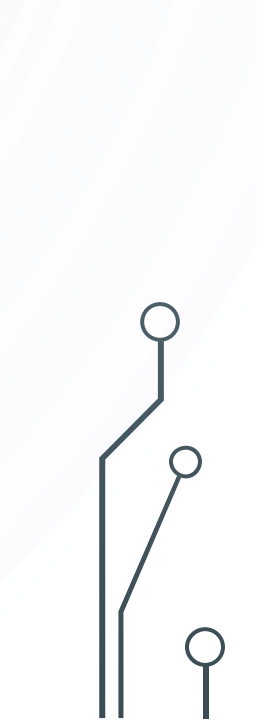
# ALLOCAZIONE IN MEMORIA DI UN RECORD

- Il compilatore alloca memoria sufficiente per accogliere tutti i campi di una struttura
- Esempio: record di 92 byte

```
struct Calciatore{  
    char nome[30];  
    char cognome[40];  
    char ruolo[20];  
    int eta;  
};
```



# ESERCIZI

- Realizzare un programma che permetta di effettuare le operazioni fondamentali sui numeri complessi (somma, sottrazione, prodotto, divisione)
    - I numeri complessi devono essere definiti mediante l'uso di record
- 
- 
- 

# OPERAZIONI SUI VETTORI: PROBLEMI DI RICERCA





# RICERCA IN UN VETTORE

- Tra le operazioni più diffuse in programmazione è la ricerca di un elemento in un dato vettore (se esiste)
- L'algoritmo più semplice e intuitivo di ricerca è la **ricerca sequenziale**
  - Si scorre il vettore dall'inizio alla fine, fintanto che non viene individuato l'elemento cercato
  - Questo è anche l'unico algoritmo possibile su un vettore non ordinato
- **Se il vettore è ordinato** si può usare un algoritmo più efficiente: la **ricerca binaria** (o dicotomica)

# RICERCA BINARIA

Valore Cercato: 10

 Primo

 Ultimo

Passo1

1	3	5	10	15	23	31	47	47	56	64
0	1	2	3	4	5	6	7	8	9	10

↑  
Elemento Centrale

Passo2

1	3	5	10	15	23	31	47	47	56	64
0	1	2	3	4	5	6	7	8	9	10

↑  
Elemento Centrale

Passo3

1	3	5	10	15	23	31	47	47	56	64
0	1	2	3	4	5	6	7	8	9	10

↑  
Elemento Centrale

# RICERCA BINARIA

*Se il valore cercato non è all'interno del vettore...*

Valore Cercato: 8

 Primo

 Ultimo

Passo4

1	3	5	10	15	23	31	47	47	56	64
0	1	2	3	4	5	6	7	8	9	10

↑  
Elemento Centrale

# COMPLESSITA' COMPUTAZIONALE: RICERCA SEQUENZIALE

- **Best case  $O(1)$ :** ricerca con successo al primo caso (primo elemento del vettore), con un solo confronto
- **Worst case  $O(N)$ :** ricerca senza successo, l'algoritmo dovrà scorrere tutto l'array di dimensione  $N$ , quindi farà  $N$  confronti
- **Medium case  $O(N)$ :** mediamente l'algoritmo effettua  $(N+1)/2$  confronti

# COMPLESSITA' COMPUTAZIONALE: RICERCA BINARIA

- **Best case  $O(1)$ :** ricerca con successo al primo caso (elemento centrale del vettore), con un solo confronto
- **Worst case  $O(\log_2(N))$ :** ricerca senza successo; ad ogni iterazione l'insieme è dimezzato, il numero di confronti è pari a quante volte  $N$  può essere diviso per 2 fino a ridurlo a 0. In un vettore di dimensione  $N = 2^h$ , l'algoritmo deve compiere circa  $h = \log_2(N)$  passi (e quindi confronti) per la ricerca senza successo.
- **Medium case  $O(\log_2(N))$ :** nel caso medio il numero è leggermente inferiore ma proporzionale a  $\log_2(N)$

# COMPLESSITA' COMPUTAZIONALE: RICERCA BINARIA

```
int a[N]={1,5,8,10,15};
int primo=0, ultimo=N-1, medio;
bool trovato=false;

while (primo<=ultimo && !trovato)
{
    medio = (int)(primo+ultimo)/2;
    if(a[medio]==x)
        trovato=true;
    else if (x > a[medio])
        primo = medio + 1;
    else /* x< a[medio] */
        ultimo = medio - 1;
}
if(trovato)
    cout<<"numero trovato in posizione: "<<medio<<endl;
else
    cout<<"numero non trovato";
```

# ESERCIZI

- Realizzare un programma che permetta di inserire un vettore in ingresso dal lato utente di N numeri interi positivi (N massimo 20).

Il programma:

- Controlla se il vettore è ordinato e lo comunica all'utente
- chiede all'utente un numero da cercare:
- dà come risposta se il numero è presente o meno e quanti confronti sono stati effettuati, utilizzando la ricerca sequenziale
- ripete la ricerca finché l'utente non inserisce un numero negativo

# ESERCIZI

- Realizzare un programma che permetta di inserire un vettore in ingresso dal lato utente di N numeri interi positivi (N massimo 20).
  - Il programma chiederà all'utente un numero da cercare:
  - Il programma darà come risposta se il numero è presente o meno e quanti confronti sono stati effettuati, utilizzando la ricerca binaria
  - Si ripete la ricerca finchè l'utente non inserisce un numero negativo
- N.B.: si presuppone che il vettore sia ordinato



# ESERCIZI

- Realizzare un programma che permetta di generare un vettore in ingresso dal lato utente di N numeri interi positivi tra 0 e 10 pseudo-randomici. N dovrà essere minimo 20 e massimo 100 scelto dall'utente. Il programma:
  - genera un vettore somma cumulata partendo dal vettore generato. (N.B. l'elemento i-esimo del vettore è pari alla somma di tutti gli elementi da 0 a i), e lo mostra all'utente.
  - chiede all'utente un numero da cercare
  - darà come risposta se il numero è presente o meno nel vettore somma cumulata e quanti confronti sono stati effettuati, utilizzando la ricerca sequenziale e quella binaria.
  - ripete finché l'utente non inserisce un numero negativo

**DOMANDE, DUBBI, PERPLESSITÀ**

