TUTORATO 08

PROGRAMMAZIONE E LABORATORIO A.A 2025-2026


Dipartimento di Matematica e Informatica

Tutor: Dott. Dominik Miotla

Union

- Una **union** è un tipo di dato che può contenere campi di tipi diverso, come una **struct**.
- La differenza è che nelle **union** solo uno dei campi viene tenuto in memoria: **i campi sono in alternativa**.
- Tutti i campi di una union condividono lo stesso spazio in memoria.

```
union {
    int i;
    char s[6];
    float f;
}
```

Visto che i campi sono memorizzati nella stessa area di memoria, se quando si utilizza un dato si usa il campo sbagliato, si ottiene un risultato privo di senso.

```
X.i=65;
printf("%d %s %f",X.i,X.s,X.f);
strcpy(X.s,"ABCD");
printf("%d %s %f",X.i,X.s,X.f)
X.f=3.1415
printf("%d %s %f",X.i,X.s,X.f)
```

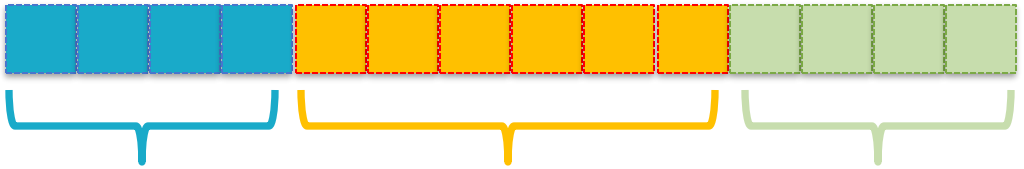
```
65          A    0
1145258561 ABCD 781.0352
1078529622 VI@  3.141500
```

Valori system-dependent!

Union vs Struct

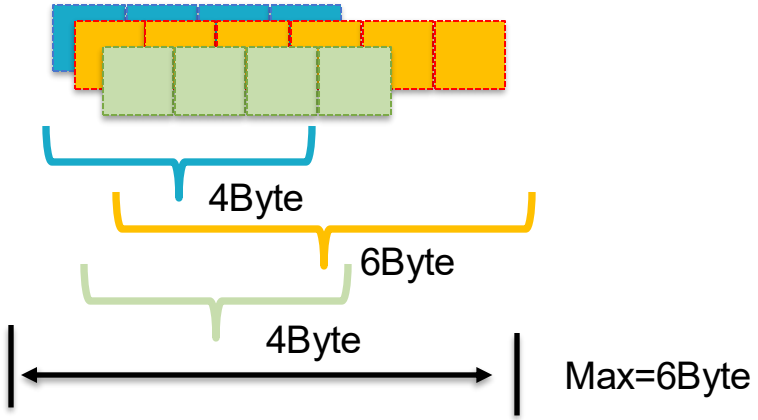
```
struct {
    int i;
    char s[6];
    float f;
}x
```

- **Tutti** e 3 i campi della struttura x sono presenti in memoria contemporaneamente.
- La struttura **contiene** un dato int**,** una stringa **e** un float.



4Byte + 6Byte + 4Byte = **14B**

```
union {
    int i;
    char s[6];
    float f;
}x
```



- **Un solo campo** della union **x** è presente in memoria in ogni momento.
- La union **può rappresentare** un dato int **oppure** una stringa **oppure** un float.

INPUT / OUTPUT CON FILE DI TESTO

- Creazione di un file pointer: `FILE* fp;`
- Apertura di un file: `FILE* fopen(char fname[], char mode[]);`
  dove **`mode`** è una *stringa* indica la modalità di apertura del file:
  - **`r`** → sola lettura (se il file non esiste, dà **errore**)
  - **`w`** → sola scrittura, sovrascrivendo il contenuto già presente (se non esiste, lo crea)
  - **`a`** → sola scrittura, aggiungendo al contenuto già presente (se non esiste, lo crea)

  e può eventualmente essere seguito da:
  - **`t`** se il file è testuale (default)
  - **`b`** se il file è binario (obbligatorio)
  - **`+`** per aprire in lettura e scrittura

INPUT / OUTPUT CON FILE DI TESTO

- **NB**: se il file aperto in lettura non esiste, fopen ritorna NULL ⇒ è buona norma mettere un controllo sull'apertura:

```
if(fp == NULL) { printf("Errore nell'apertura del file."); }
```

- Ricordarsi sempre di chiudere il file con

```
fclose(FILE* fp);
```

che ritorna 0 se è andato tutto bene.

FILE DI TESTO
LETTURA

```
FILE* fp = fopen(fileName, “rt”);
```

- Per leggere *tutto* il contenuto di un file si usa

```
int fscanf(fp, FORMAT, DEST);
```

- **FORMAT** è una stringa che stabilisce cosa verrà letto con lo specificatore di formato (e.g., “%d”, “%c”, “%s”,… anche più di uno)
- **DEST** è **indirizzo di memoria** della variabile dentro cui salvare il contenuto letto
- **fp** è il file da cui leggere
- Restituisce il numero di caratteri/stringhe/interi/… letti con successo, altrimenti un numero negativo

Esempio:

```
int eta; char cognome [30];
int val = fscanf(fp,"%s %d", cognome, &eta);
```

legge 2 elementi

FILE DI TESTO
SCRITTURA

```
FILE* fp = fopen(fileName, “wt”);
```

- Per scrivere un file si usa

```
int fprintf(fp, FORMAT, ARG1, ARG2, ...);
```

- **FORMAT** è una stringa che specifica cosa verrà scritto con lo specificatore di formato (e.g., “%d”, “%c”, “%s”,… anche più di uno: “%d %s %c”)
- **ARG1,ARG2,...** sono le variabili il cui contenuto verrà scritto sul file
- **fp** è il file su cui scrivere
- Restituisce il numero di caratteri/stringhe/interi/… scritti con successo, altrimenti un numero negativo

Esempio:

```
int val = fprintf(fp, "%s %d", “Pippo”, 50);
```

INPUT / OUTPUT CON FILE **BINARI**

- Creazione di un file pointer: `FILE* fptr;`
- Apertura di un file: `FILE* fopen(char fname[], char mode[]);`
  dove **`mode`** è una *stringa* indica la modalità di apertura del file:
  - **`r`** → sola lettura (se il file non esiste, dà **errore**)
  - **`w`** → sola scrittura, sovrascrivendo il contenuto già presente (se non esiste, lo crea)
  - **`a`** → sola scrittura, aggiungendo al contenuto già presente (se non esiste, lo crea)

  e può eventualmente essere seguito da:
  - **`t`** se il file è testuale (default)
  - **`b`** se il file è binario (obbligatorio)
  - **`+`** per aprire in lettura e scrittura

FILE BINARI
LETTURA

```
FILE* fp = fopen(fileName, "rb");
```

- Per leggere il contenuto (sequenza di byte!) di un file binario si usa

```
int fread(DEST, int dimElem, int numElem, FILE *f);
```

- **DEST**: **indirizzo di memoria** della variabile in cui salvare gli elementi letti
- **dimElem**: dimensione *in byte* del dato da leggere
- **numElem**: numero di elementi che si vogliono leggere alla volta, ognuno di `dimElem`
- **f** è il file da cui leggere

Il valore di ritorno è il numero di elementi *completi* letti dalla funzione ⇒ se minore da `numElem`, significa che qualcosa è andato storto

Esempio:

```
typedef struct{ char nome[30], cognome[30]; int eta; } persona;
persona prsn;
int val = fread(&prsn, sizeof(persona), 1, fp);
```

FILE BINARI
SCRITTURA


```
FILE* fp = fopen(fileName, "wb");
```

- Per scrivere (sequenza di byte!) su un file binario si usa

```
int fwrite(SOURCE, int dimElem, int numElem, FILE *fp);
```

- **SOURCE** indirizzo di memoria della variabile il cui contenuto verrà scritto sul file
- **dimElem**: dimensione *in byte* del dato da scrivere
- **numElem**: numero di elementi che si vogliono scrivere alla volta, ognuno di `dimElem`
- **f** è il file su cui scrivere
- Restituisce il numero di elementi scritti con successo ⇒ se minore da `numElem`, significa che qualcosa è andato storto

Esempio:

```
typedef struct{ char nome[30], cognome[30]; int eta; } persona;
persona prsn = {"Mario", "Rossi", 50};
int val = fwrite(&prsn, sizeof(persona), 1, fp);
```

ABSTRACT DATA TYPE (ADT)

Solitamente abbiamo a che fare con strutture di dati del tipo:

```
typedef struct
{     Automobile arr [DIM];
      int dl;
} Autosalone;
```

Un ADT definisce cosa un tipo di dato può fare (le sue operazioni) e non come lo fa (l'implementazione interna), cosa vuol dire?

In poche parole NON devo modificare direttamente i campi arr , dl della struttura Autosalone nel main!

ABSTRACT DATA TYPE (ADT)

```
typedef struct
{     Automobile arr [DIM];
      int dl;
} Autosalone;

int main() {

Autosalone au;
inizializza(&au);
aggiungi_auto(&au);
stampa(&au);
..
}
```

Nel main **NON** sto modificando direttamente Autosalone!

Quello che posso fare è richiamare delle funzioni che modificheranno esternamente il contenuto di Autosalone:

- inizializza()
- aggiungi_auto()
- stampa()

Se un giorno dovessi modificare un campo di Autombile, il main rimarrebbe invariato, dovrei solo gestire le funzioni che utilizzano questo dato.

ESERCIZIO 1: NOMI ORDINATI

Scrivere un programma che:

- legge i dati di alcuni studenti da un file di testo “nomi.txt”
  - per ogni riga, il file contiene: cognome, nome, matricola
  - nome e cognome sono stringhe di massimo 20 char
- li salva all’interno di un array
- ordina l’array in base al numero di matricola
- salva i dati ordinati all’interno di un file di testo “ordinati.txt”

ESERCIZIO 1: SECONDA PARTE

Aggiungere una funzione all'esercizio precedente che scriva i dati ordinati nel file "ordinati.dat"

- Scrivere un nuovo programma che legga il file "ordinati.dat"
- Stampi a video il contenuto
- Scriva nel file "primo.txt" la prima riga del file ordinato "ordinati.dat"

ESERCIZIO 2: RUBRICA v2

Modificare il programma che simula una rubrica in modo tale che:

- legga i contatti inseriti da tastiera dall'utente (anche il numero di contatti da leggere viene chiesto in input all'utente)
- stampi i contatti ordinati per cognome in un file di testo rubrica.txt

Modificare opportunamente le funzioni della versione originale dove necessario.

ESERCIZIO 3 : SIMULAZIONE ESAME

Svolgere la parte 1 del seguente tema d'esame : link

Suggerimento: definire le seguenti strutture …

```
#define MAX_MEMORIE 100
#define DIM_COSTRUTTORE 20

typedef struct {
    char costruttore[DIM_COSTRUTTORE];
    int tempo_accesso; // in ns
    int costo;         // in $
} Memoria;

typedef struct {
    Memoria memorie[MAX_MEMORIE];
    int indice; // Numero di memorie attuali
} Registro;
```

ricordarsi della questione ADT …

```
// Funzioni per la gestione del Registro (ADT)
void inizializza_registro(Registro *r);
int aggiungi_memoria(Registro *r, Memoria m);
void stampa_memorie(Registro *r);

...
```