



Programmazione 1 (Canale O-Z)

Lezione 13: Modellare la Realtà (Struct, Padding e Persistenza a Blocchi)

Docente: Prof. Antonino Furnari

Email: `antonino.furnari@unict.it`

Sito Web: antoninofurnari.github.io/prog1-2627

🧩 La Realtà è Eterogenea

- Finora: **array** (dati omogenei) e **file** di testo e binari.
- Un'entità reale come uno **Studiante** ha caratteristiche di tipo diverso: nome (stringa), matricola (`int`), media (`float`).
- Oggi:
 - `struct`: aggregare dati eterogenei in un **record**;
 - `.` e `→`, padding, `enum`;
 - `struct` + file binari: I/O a blocchi e `fseek`.



! Array Paralleli

Per gestire un elenco di esami (materia, CFU, voto) finora usavamo **tre array** che condividono lo stesso indice logico:

```
char materie[100][50]; // Nomi degli esami
int cfu[100];           // Crediti di ciascun esame
int voti[100];          // Voto conseguito
```

- L'esame `i` è "sparso" tra `materie[i]`, `cfu[i]` e `voti[i]`.

? Vogliamo ordinare gli esami per voto con il Bubble Sort: cosa succede quando scambiamo l'esame `i` con l'esame `j`?

La Fragilità degli Array Paralleli

```
// Scambio del voto
int temp_voto = voti[i]; voti[i] = voti[j]; voti[j] = temp_voto;

// Scambio dei CFU
int temp_cfu = cfu[i]; cfu[i] = cfu[j]; cfu[j] = temp_cfu;

// Scambio della stringa materia
char temp_mat[50];
strcpy(temp_mat, materie[i]);
strcpy(materie[i], materie[j]);
strcpy(materie[j], temp_mat);
```

- **Tre swap** per un solo scambio logico (e con 10 campi?).
- Basta dimenticarne **uno** per attribuire i CFU di Matematica all'esame di Storia.
- Il C ha `int`, `float`, `char`... ma **non** un tipo `Esame`: dobbiamo crearlo noi.



Un Nuovo Tipo: la **struct**

```
struct materia_s {  
    char nome[50];  
    char codice[10];  
    int cfu;  
    int voto;  
}; // ATTENZIONE: Il punto e virgola finale è obbligatorio!
```

- **struct** raggruppa variabili di tipo diverso in un unico contenitore **logico e fisico**.
- Ogni variabile interna è un **campo** (o **membro**).
- La definizione è solo il **progetto** (lo "stampo"): **non alloca RAM**.

L'Operatore Punto (.)

```
#include <stdio.h>
#include <string.h>

int main(void) {
    struct materia_s analisi;           // ripetiamo la keyword 'struct'

    strcpy(analisi.nome, "Analisi");    // i campi array si riempiono con strcpy
    strcpy(analisi.codice, "AN29345");
    analisi.cfu = 9;
    analisi.voto = 28;

    printf("Voto in analisi: %d\n", analisi.voto);
    return 0;
}
```

Voto in analisi: 28

 **Live:** gcc -Wall analisi.c -o analisi && ./analisi. Poi provare `analisi.nome = "Analisi";` e leggere l'errore di gcc (un array non si assegna con `=`).

✨ typedef : un Nome per il Nuovo Tipo

Forma compatta (quella che useremo):

```
typedef struct {  
    char nome[50];  
    char codice[10];  
    int cfu;  
    int voto;  
} Esame; // Esame è un nuovo tipo!  
  
Esame e1; // niente 'struct'
```

Forma scorporata:

```
struct esame {  
    char nome[50];  
    char codice[10];  
    int cfu;  
    int voto;  
};  
  
typedef struct esame Esame;
```

- `typedef` crea un **alias** (sinonimo) di un tipo esistente.
- **PascalCase** per i tipi nostri (`Esame`, `StudenteIngegneria`): si distinguono a colpo d'occhio da `int`, `float`, `char`.



Inizializzazione a Graffe

```
// Corretto: {nome, codice, cfu, voto}  
Esame prog = {"Programmazione 1", "INF101", 12, 30};
```

```
// Errato o disallineato:  
Esame errore = {"Fisica", "FIS01", 18, 6}; // Assegna 18 a cfu e 6 a voto!
```

- I valori vengono assegnati **nell'ordine di dichiarazione** dei campi.
- Il compilatore non può sapere che 18 era il voto: i tipi coincidono, **nessun errore**.

? Con gli array `int b[3] = a;` non compila. E con le struct?

Copia per Assegnamento

```
Esame backup = prog; // Copia immediata di tutti i campi in RAM
```

- A differenza degli array, le struct si copiano con `=`.
- Il compilatore effettua una copia **bit-a-bit** dell'intera area di memoria della struct.
- **Anche gli array contenuti** nella struct (es. `nome[50]`) vengono copiati.

 **Live:** dopo `backup = prog;` modificare `backup.voto` e stampare entrambi: `prog.voto` resta 30 (sono due copie distinte).



Passaggio per Valore: Quanto Pesa?

$50 \text{ (nome)} + 10 \text{ (codice)} + 4 \text{ (cfu)} + 4 \text{ (voto)} = 68 \text{ byte}$

```
void stampa_copia(Esame e) {  
    printf("%s: %d\n", e.nome, e.voto);  
}
```

- La chiamata `stampa_copia(prog)` **duplica tutti i 68 byte** nel record di attivazione sullo Stack.
- Con array più grandi nella struct: spreco di Stack e cicli di clock per la "fotocopia".

? Come possiamo evitare la copia? Cosa abbiamo imparato a passare al posto del dato stesso?

⚙️ Passaggio per Riferimento: l'Operatore →

Passiamo solo l'indirizzo della struct (8 byte su sistemi a 64-bit):

```
void stampa_rif(const Esame *e) {  
    // 1. Sintassi esplicita: servono le parentesi (precedenza di . su *)  
    printf("%s: %d\n", (*e).nome, (*e).voto);  
  
    // 2. Operatore freccia: zucchero sintattico per (*e).campo  
    printf("%s (%s) - CFU: %d, Voto: %d\n", e→nome, e→codice, e→cfu, e→voto);  
}  
  
int main(void) {  
    Esame info = {"Informatica", "INF01", 6, 27};  
    stampa_rif(&info); // Passiamo l'indirizzo  
    return 0;  
}
```

```
Informatica: 27  
Informatica (INF01) - CFU: 6, Voto: 27
```



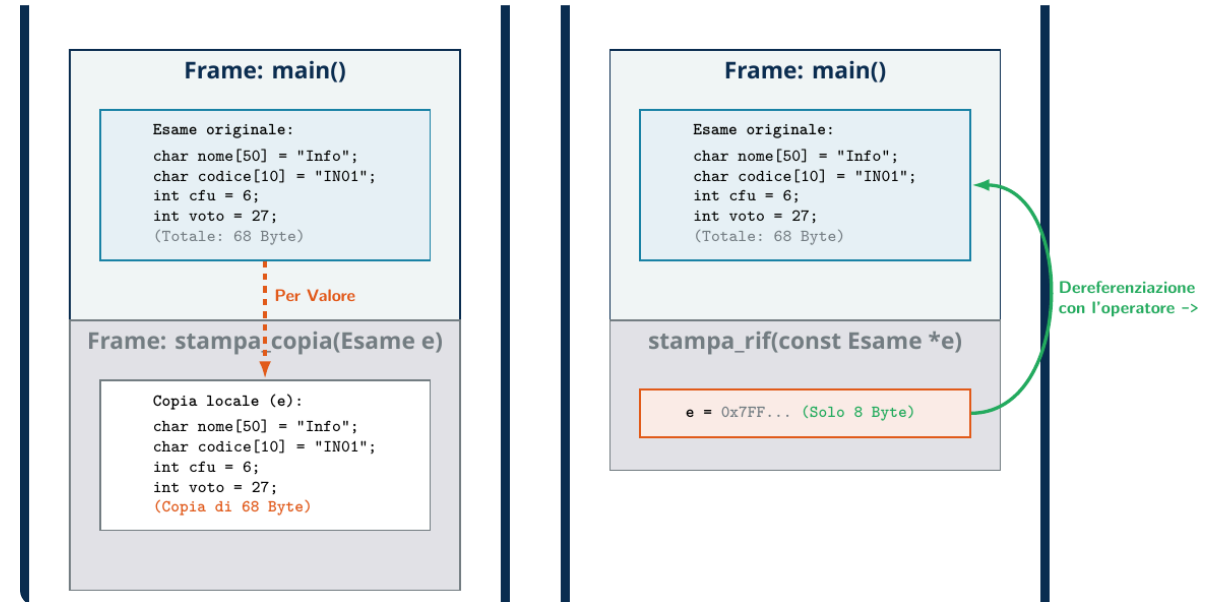
Valore vs Riferimento in Memoria

- **Per valore** `void stampa(Esame e)`: copia di 68 byte sullo Stack.
- **Per riferimento** `void stampa(const Esame *e)`: solo 8 byte (l'indirizzo).
- `const`: la funzione **non può modificare** la struct puntata.
- **Regola pratica**: hai la struct ➡ `.`; hai il puntatore alla struct ➡ ➔.



Live: eseguire l'esempio su Python Tutor e confrontare i frame di `stampa_copia` e `stampa_rif`.

Scenario A: Passaggio per Valore Scenario B: Passaggio per Riferimento





Il Mistero delle Dimensioni

```
typedef struct {  
    char c;      // 1 byte  
    double d;    // 8 byte  
    int i;       // 4 byte  
} ProvaA;
```

```
typedef struct {  
    double d;    // 8 byte  
    int i;      // 4 byte  
    char c;     // 1 byte  
} ProvaB;
```

- Stessi campi, ordine diverso. Algebricamente: $1 + 8 + 4 = 13$ byte.

? Quanto valgono `sizeof(ProvaA)` e `sizeof(ProvaB)` ? Scommettiamo prima di eseguire.



Il Verdetto di `sizeof`

```
printf("ProvaA: %zu byte\n", sizeof(ProvaA));  
printf("ProvaB: %zu byte\n", sizeof(ProvaB));
```

```
ProvaA: 24 byte  
ProvaB: 16 byte
```

- **Nessuna delle due** occupa 13 byte, e l'ordine dei campi **conta!**

 **Live:** `gcc -Wall padding.c -o padding && ./padding` (su una macchina a 64-bit).



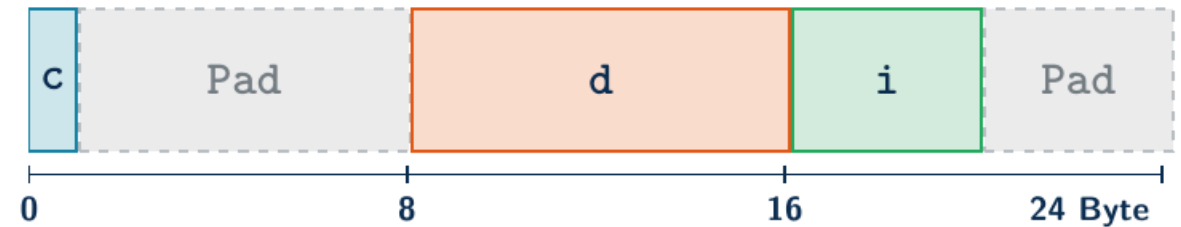
Il Padding: Perché Accade

- La CPU legge la RAM a **blocchi allineati** (di solito 8 byte su sistemi a 64-bit).
- Un `double` a cavallo tra due blocchi richiederebbe **due letture** RAM ➡ più lento.
- Regole di allineamento applicate dal compilatore:
 - i. Ogni tipo nativo inizia a un indirizzo **multiplo della sua dimensione** (`double` ➡ multipli di 8, `int` ➡ multipli di 4).
 - ii. Si iniettano byte vuoti invisibili (**padding**) per ripristinare l'allineamento.
 - iii. La dimensione totale è multipla di quella del campo più grande.

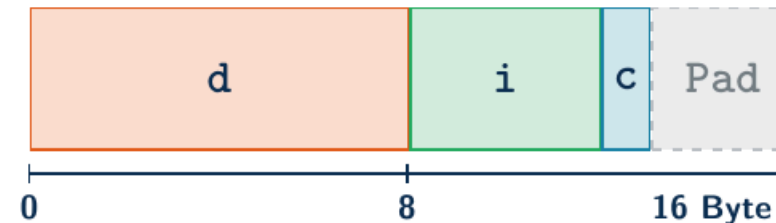
Visualizzazione del Padding in Memoria

- **ProvaA (24 Byte):**
 - **c** all'offset 0, poi **7 byte** di padding.
 - **d** dall'offset 8 al 15.
 - **i** dall'offset 16 al 19, poi **4 byte** finali (multiplo di 8).
- **ProvaB (16 Byte):**
 - **d** da 0 a 7, **i** da 8 a 11, **c** all'offset 12.
 - Solo **3 byte** di padding finali.
- Ordinare i campi dal più grande al più piccolo **riduce lo spreco**.

struct ProvaA (Dimensione Totale = 24 Byte)



struct ProvaB (Dimensione Totale = 16 Byte)



! Padding: Quando Preoccuparsene

- Nei normali esercizi di laboratorio **non** è richiesto riordinare le struct per risparmiare memoria: disponiamo i campi nel modo più **logico e leggibile**.
- Conoscere il padding è però fondamentale a basso livello:
 - sistemi operativi;
 - sistemi embedded (microcontrollori), dove ogni byte conta;
 - programmazione di rete.
- **Morale:** `sizeof` di una struct si **chiede** al compilatore, non si calcola a mano.

? Un esame può essere "In Corso", "Superato" o "Bocciato". Se lo salviamo come `0`, `1`, `2`, chi legge il codice capisce cosa significa `stato = 2`?

Enumerazioni: `enum`

```
typedef enum {  
    IN_CORSO,    // 0  
    SUPERATO,    // 1  
    BOCCIATO     // 2  
} StatoEsame;
```

- Associano **nomi mnemonici** a costanti intere incrementali (di default da 0).
- Sotto il cofano sono normali interi, ma eliminano i "**numeri magici**" sparsi nel codice.
- `StatoEsame` è un nuovo tipo: possiamo usarlo come campo di una struct.



enum dentro una struct

```
typedef struct {
    char materia[50];
    StatoEsame stato;
} EsameCarriera;

int main(void) {
    EsameCarriera prog = {"Programmazione 1", IN_CORSO};

    if (prog.stato == IN_CORSO) {
        printf("Stai ancora studiando %s. Coraggio!\n", prog.materia);
    }

    prog.stato = SUPERATO; // Aggiornamento dello stato
    return 0;
}
```

Stai ancora studiando Programmazione 1. Coraggio!

- Il codice si legge "come un libro": `prog.stato == IN_CORSO`.

Struct e File Binari

- **Lezione 12:** file binari con stringhe a **lunghezza variabile** ➡ impossibile calcolare l'offset di un record.
- La struct è il **mattoncino a dimensione fissa**: `sizeof(Esame)` è nota a tempo di compilazione.
- Un file binario fatto di struct in sequenza si comporta come un **array su disco**.
- Anche in RAM un array di struct è un **blocco contiguo**: si può scrivere tutto in un colpo.

? Se l'array `libretto` è contiguo in RAM, quante chiamate a `fwrite` servono per salvarlo tutto?

Salvataggio dell'Inter Database

```
#include <stdio.h>

typedef struct {
    char nome[50];
    int cfu;
    int voto;
} Esame;

int main(void) {
    Esame libretto[3] = {
        {"Analisi 1", 9, 28},
        {"Programmazione 1", 12, 30},
        {"Fisica 1", 6, 25}
    };

    FILE *fp = fopen("database.bin", "wb");
    if (fp != NULL) {
        // Scrittura in blocco: (sizeof(Esame) * 3) byte trasferiti sul disco
        fwrite(libretto, sizeof(Esame), 3, fp);
        fclose(fp);
        printf("Database salvato con successo!\n");
    }
    return 0;
}
```

- Una **sola** `fwrite` per tutti i record. Nota: questa `Esame` **non** ha il campo `codice`.

Verifichiamo il File su Disco

```
gcc -Wall salva.c -o salva && ./salva
```

Database salvato con successo!

- Il file contiene `3 * sizeof(Esame)` byte: la dimensione **esatta** dipende dal padding.

 **Live:** aggiungere `printf("%zu\n", sizeof(Esame));`, poi `ls -l database.bin` e confrontare con 3 volte quel valore. Aprire il file con `xxd database.bin | head`: i nomi sono leggibili, i numeri no (sono in binario).



Accesso Diretto al Record N-esimo

- Blocchi di dimensione costante ➡ la testina salta **subito** all'inizio del record N -esimo.
- Offset in byte dall'inizio del file (`SEEK_SET`):
$$\text{Offset} = \text{indice} \times \text{sizeof}(\text{Struct})$$
- Nessuna scansione dei record precedenti, nessun caricamento in RAM del resto del file.

File Binario database.bin (Blocchi da 68 Byte)

```
fseek(fp, 1 * sizeof(Esame), SEEK_SET);
```



Accesso Diretto: Codice C

```
int main(void) {  
    FILE *fp = fopen("database.bin", "rb");  
    if (fp == NULL) {  
        return 1;  
    }  
  
    Esame cercato;  
    int indice_desiderato = 1; // Vogliamo leggere il secondo esame (indice 1)  
  
    // SALTO MATEMATICO  
    fseek(fp, indice_desiderato * sizeof(Esame), SEEK_SET);  
  
    // Lettura isolata dell'unico record posizionato sotto la testina  
    fread(&cercato, sizeof(Esame), 1, fp);  
    fclose(fp);  
  
    printf("Esame estratto al volo: %s, Voto: %d\n", cercato.nome, cercato.voto);  
    return 0;  
}
```

Esame estratto al volo: Programmazione 1, Voto: 30

 **Live:** gcc -Wall leggi.c -o leggi && ./leggi (stessa Esame del salvataggio!), poi provare indice_desiderato 0 e 2.

✓ Cosa Abbiamo Imparato Oggi

- Gli **array paralleli** sono fragili: la `struct` aggrega dati eterogenei in un unico record.
- `typedef` crea un alias (PascalCase); inizializzazione a graffe **in ordine**; copia con `=`.
- Hai la struct  `.`; hai il puntatore  `→`. Passare `const Esame *` evita la copia sullo Stack.
- **Padding**: il compilatore allinea i campi, `sizeof` dipende dall'ordine.
- `enum`: nomi mnemonici al posto dei numeri magici.
- Struct + file binari: `fwrite` in blocco e `fseek(fp, i * sizeof(T), SEEK_SET)`.

 **Per casa:** esercizi della Lezione 13 sul sito del corso.

Grazie per l'attenzione!

Sito Web del Corso: antoninofurnari.github.io/prog1-2627