Appunti Esame Orale — Laboratorio 2

Parte I — Fondamenta del C

Tema 1: Compilazione, Shell e Tipi di Base

Il linguaggio C è compilato

Il C viene tradotto direttamente in **linguaggio macchina** (binario) dal compilatore (`gcc`). A differenza di Python, gli errori di sintassi bloccano la creazione dell'eseguibile.

> *[!WARNING]*
> *Gli errori **logici** e di **gestione della memoria** (es. buffer overflow) **non vengono rilevati** dal compilatore: rimangono nel codice compilato e causano i famosi **Segmentation Fault** durante l'esecuzione.*

Il compilatore `gcc` e `-Wall`

Il flag `-Wall` abilita tutti i **Warning** (avvisi) principali.

Tipo	Comportamento
Errore	Es. manca un `;` **ferma** il compilatore, non crea il file eseguibile.
Warning	Es. variabile dichiarata ma non usata segnala il sospetto, **ma compila**.

> *[!TIP]*
> *Se il prof te lo chiede: "Il vero programmatore C scrive codice che compila senza alcun warning."*

Comandi Shell utili

Comando	Descrizione

`cat file.txt`	Stampa tutto il contenuto a terminale (scomodo per file lunghi).
`less file.txt`	Visualizzatore interattivo: scorrimento su/giù.
`./programma > output.txt`	Ridirige **stdout** (fd 1) su un file (sovrascrive).
`./programma 2> errori.txt`	Ridirige **stderr** (fd 2) su un file.
`./programma > output.txt 2> err.txt`	Separa output ed errori in due file distinti.

Tipi e Memoria

Tipo	Dimensione tipica	Note
`int`	**4 byte** (32 bit)	—
Puntatore (`char *` , `int *`)	**8 byte** (64 bit)	Contiene un **indirizzo di memoria**, non un dato diretto.

Tema 2: Le Stringhe in C e il tipo `void *`

> *[!IMPORTANT]*
> *Domanda d'esame!*

Non esiste il tipo "stringa" nativo in C. Una stringa è un **array di caratteri** identificato da un puntatore al primo elemento (`char *`).

Il Terminatore `\0`

Ciò che distingue un semplice array di `char` da una vera stringa è il **terminatore Null** (`\0` , byte `0`). Tutte le funzioni di libreria (`printf` , `strlen` , ecc.) scorrono la memoria byte per byte e si fermano **solo** quando incontrano `\0` .

> *[!CAUTION]*
> *Se dimentichi il terminatore, `printf` continuerà a leggere la RAM stampando "spazzatura" fino al crash (Segmentation Fault).*

Il tipo `void *`

`void *` è un **puntatore generico**: *"punto a un indirizzo di memoria, ma non so che tipo di dato ci sia dentro"*.

- Usato da funzioni come `malloc` (restituisce blocchi di byte puri) e `qsort`.
- **Non può essere dereferenziato** direttamente: prima devi fare un cast al tipo corretto.

```
int *p = (int *)malloc(10 * sizeof(int));  // Cast da void* a int*
```

Tema 3: Gestione della Memoria (Stack vs Heap)

> *[!IMPORTANT]*
> *Questo è il **cuore dell'esame**.*

Aspetto	Stack (Statico)	Heap (Dinamico)
Dichiarazione	`int arr[10];`	`int *arr = malloc(10 * sizeof(int));`
Allocazione	Automatica (entrando nella funzione)	Manuale (`malloc`, `calloc`, `realloc`)
Distruzione	Automatica (uscendo dalla funzione)	Manuale (`free`) — **obbligo del programmatore**
Dimensione	Fissa (nota a compile-time)	Variabile (decidibile a runtime)

Le funzioni di allocazione — `<stdlib.h>`

3.1 `malloc`

```
void *malloc(size_t size);
```

Parametro	Tipo	Descrizione
Ritorno	`void *`	Puntatore all'area allocata. **`NULL`** se l'Heap è pieno.
`size`	`size_t`	Numero totale di byte da allocare.

> *[!WARNING]*
> *È **obbligatorio** controllare sempre `if (ptr == NULL)` prima di usare il puntatore!*

3.2 `calloc`

```
void *calloc(size_t nmemb, size_t size);
```

Parametro	Tipo	Descrizione
Ritorno	`void *`	Puntatore all'area allocata e **azzerata**. `NULL` in errore.
`nmemb`	`size_t`	Numero di elementi.
`size`	`size_t`	Dimensione del singolo elemento.

> *[!NOTE]*
> ***Differenza con `malloc` :*** *`calloc` **pulisce la memoria** (tutti i bit a 0). `malloc` è più veloce ma lascia dentro la "spazzatura" (dati residui di vecchi programmi).*

3.3 `realloc` (Molto importante!)

```
void *realloc(void *ptr, size_t size);
```

Serve a **ridimensionare** un blocco di memoria dinamica esistente. All'esame devi descrivere i **due scenari**:

Scenario	Cosa succede
C'è spazio contiguo libero	Il kernel allarga il blocco in fondo. Il puntatore ritornato **rimane uguale** a `ptr`.
Non c'è spazio (Frammentazione)	1. Cerca un nuovo blocco abbastanza grande. 2. **Copia** bit per bit i vecchi dati. 3. Fa `free` del vecchio blocco. 4. Restituisce il **nuovo puntatore**.

```
ptr = realloc(ptr, nuova_size);  // SEMPRE riassegnare il puntatore!
```

3.4 `free` (Dettaglio da Esame)

```
void free(void *ptr);
```

Si usa **esclusivamente** per distruggere memoria creata con `malloc` / `calloc` / `realloc`. **Mai** usare `free` su variabili statiche dello Stack!

> *[!CAUTION]*
> ***Dangling Pointer:*** *`free(a)` **non cancella i dati** in memoria, né trasforma `a` in `NULL`. Si limita a comunicare al gestore della memoria che quei byte sono di nuovo liberi. Se per errore fai `a[0] = 5;` dopo la `free`, il programma compilerà e magari non andrà in crash subito, ma **corromperai in modo invisibile** la memoria del programma.*

Tema 4: Input/Output e Gestione Errori

`scanf` — `man 3 scanf`

```
#include <stdio.h>
int scanf(const char *format, ...);
```

Parametro	Tipo	Descrizione
Ritorno	`int`	Numero di argomenti convertiti con successo. Può essere **minore del previsto** o `EOF`.
`format`	`const char *`	Stringa di formato (es. `"%d"` per intero).
`...`		Puntatori alle variabili dove salvare i valori (es. `&n`).

> *[!NOTE]*
> *Se l'utente inserisce `"ciao"` quando ci si aspetta un numero, `scanf` restituisce `0` (zero argomenti convertiti). Se si raggiunge la fine dell'input restituisce `EOF`.*

`perror` — `man 3 perror`

```
#include <stdio.h>
void perror(const char *s);
```

Aspetto	Dettaglio

Dove stampa?	Su **stderr** (non stdout!) — non confonde errori con output.
Come funziona?	Stampa `s`, seguito da `:`, poi il messaggio di sistema corrispondente a `errno`.
Esempio d'esame	`perror("Errore file")` `Errore file: No such file or directory`

Apertura File: modalità `"w"` vs `"a"`

Modalità	Comportamento
`"w"`	Se il file esiste, lo **svuota** (tronca a 0 byte) e lo ricrea.
`"a"`	**Mantiene** il contenuto, sposta il cursore alla fine aggiunge dati in coda.
`"r"`	Solo lettura. Il file **deve già esistere**, altrimenti restituisce `NULL`.

La `fclose()` e i Buffer (Punto vitale!)

Le funzioni di libreria (`fprintf`, `fwrite`) **non scrivono subito sul disco**, ma in un **buffer** (array temporaneo in RAM). La scrittura fisica avviene solo quando il buffer si riempie.

> *[!IMPORTANT]*
> ***Domanda d'esame: Perché chiamiamo sempre `fclose(file)`?*** *Oltre a liberare il File Descriptor, `fclose` esegue forzatamente il **"Flush"** del buffer: prende gli ultimi byte rimasti in RAM e li scrive su disco. Se il programma va in crash prima della `fclose`, il file potrebbe risultare **scritto a metà**.*

Tema 5: Prototipi "Vecchi" in C (Dichiarazione Implicita)

Problema classico del C pre-standard (C89):

Se chiami nel `main` una funzione (es. `calcola(5)`) **senza** aver messo il prototipo in cima al file, il vecchio compilatore C:

1. **Non va in errore** — prova a "indovinare"
2. Assume di default che `calcola` ritorni `int` e prenda parametri non specificati
3. Quando poi trova la vera definizione (es. `void calcola(float n)`), dà **"Conflitto di tipi"**

> *[!NOTE]*
> *Nelle versioni moderne di* `gcc` *, questo comportamento è diventato un* ***Errore fatale*** *e il programma non compila. Metti* ***sempre*** *i prototipi in cima al file o in un file header (* `.h` *).*

Tema 6: La Trappola della Memoria — Ritornare Array dalle Funzioni

Questo è l'errore classico in C: il **Dangling Pointer**.

Il problema (Stack)

Se dentro una funzione crei un array statico (`char buffer[50];`) e fai `return buffer;` , il compilatore potrebbe darti un warning ma te lo lascia fare.

Il problema: `buffer` vive nello **Stack Frame** di quella funzione. Appena la funzione termina, lo Stack Frame viene **distrutto/invalidato**. Il puntatore ritornato punterà a una zona di memoria "libera", che verrà presto sovrascritta **Segmentation Fault**.

La soluzione (Heap)

Se devi ritornare un array/stringa, allocalo con `malloc` . La memoria nell'Heap **sopravvive** alla fine della funzione e rimane valida finché non fai `free` .

> *[!TIP]*
> ***Perché nel*** `main` ***posso usare array statici?*** *Perché il* `main` *è la base dello Stack. Termina solo quando termina l'intero programma.*

Tema 7: Anatomia delle Stringhe e Modificabilità

Caratteri come Interi

In C non esiste un vero "tipo carattere". Un `char` è un **intero di 1 byte** (da -128 a 127). Quando stampi con `%c` , `printf` guarda la tabella **ASCII** (es. `65` `'A'`).

Stringhe letterali (Immutabili)

```
char *str = "ciao";  // "ciao" è in una sezione di memoria READ-ONLY
str[0] = 'm';        //    CRASH IMMEDIATO!
```

> *[!CAUTION]*
> *Le stringhe letterali vengono salvate dal compilatore in una sezione di memoria* ***Read-Only****. Tentare di modificarle causa un crash.*

Il parametro `const char *`

Lo trovi in quasi tutti i prototipi del `man`. Significa: *"Io funzione ti prometto che leggerò la tua stringa, ma NON la modificherò."*

`size_t`

È un intero **senza segno** (`unsigned long`), usato sempre per dimensioni in byte. Non ha senso una memoria di dimensione negativa.

Tema 8: Copiare e Confrontare Stringhe

> *[!IMPORTANT]*
> ***Domanda d'esame!*** *Il prof chiede esplicitamente la differenza tra* `strcpy` *e* `strdup` *.*
> *Entrambe in* `<string.h>` *.*

8.1 `strcpy` — String Copy

```
char *strcpy(char *dest, const char *src);
```

Parametro	Tipo	Descrizione
Ritorno	`char *`	Puntatore a `dest` .
`dest`	`char *`	Buffer di destinazione (**devi averlo già allocato**).
`src`	`const char *`	Stringa sorgente da copiare.

> *[!WARNING]*
> ***Pericolo:*** `strcpy` *presuppone che* `dest` *sia* ***grande abbastanza****. Se* `src` *è lunga 100 e* `dest` *ha spazio per 10* ***Buffer Overflow****.*

8.2 `strdup` — String Duplicate (Molto usata dal prof!)

```
char *strdup(const char *s);
```

Parametro	Tipo	Descrizione
Ritorno	`char *`	Puntatore alla nuova copia (allocata con `malloc`). `NULL` in errore.
`s`	`const char *`	Stringa da duplicare.

Cosa fa internamente: `strlen(s)` `malloc(lunghezza + 1)` `strcpy` ritorna il puntatore.

> *[!IMPORTANT]*
> ***Regola d'oro (da dire all'esame):*** *Poiché* `strdup` *chiama* `malloc` ***di nascosto****, il programmatore è* ***obbligato*** *a chiamare* `free()` *sul puntatore restituito per evitare* ***memory leak****.*

8.3 `strcmp` — String Compare

```
int strcmp(const char *s1, const char *s2);
```

Valore ritornato	Significato
`0`	Le stringhe sono **identiche**.
`< 0`	`s1` viene **prima** di `s2` (alfabetico).
`> 0`	`s1` viene **dopo** `s2` (alfabetico).

> *[!CAUTION]*
> ***Mai*** *scrivere* `if (s1 == s2)` *per confrontare stringhe! Confronteresti gli* ***indirizzi di memoria****, non il testo. Usa sempre* `strcmp`*.*

Tema 9: La Famiglia `printf` e i Buffer (`sprintf`, `snprintf`, `asprintf`)

A volte vuoi “stampare” dentro una variabile stringa (es. per creare nomi di file `log1.txt`, `log2.txt`).

`sprintf` — Pericolosa

```
int sprintf(char *str, const char *format, ...);
```

Scrive il risultato formattato dentro `str`. **Rischio altissimo** di buffer overflow se la stringa generata è più lunga del buffer.

`snprintf` — Sicura

```
int snprintf(char *str, size_t size, const char *format, ...);
```

Parametro	Tipo	Descrizione
Ritorno	`int`	Numero di caratteri che *sarebbero* stati scritti (utile per controllo).
`str`	`char *`	Buffer di destinazione.
`size`	`size_t`	Dimensione massima del buffer. Scrive al massimo `size - 1` char + `\0`.

`asprintf` — Magica e Dinamica

```
int asprintf(char **strp, const char *fmt, ...);
```

Parametro	Tipo	Descrizione
Ritorno	`int`	Numero di caratteri scritti, `-1` in errore.
`strp`	`char **`	Indirizzo di un puntatore. La funzione fa `malloc` e lo aggiorna.

> *[!TIP]*
> *Non le passi un buffer: le passi `&stringa`. Lei calcola la lunghezza, fa `malloc`,*

> *scrive e aggiorna il puntatore.* ***Ricordati di fare*** `free(stringa)` ***!***

Tema 10: Array di Stringhe (`char **`)

Se `char *` è un puntatore a un carattere (una stringa), `char **` è un **puntatore a un puntatore** a un carattere nella pratica, un **Array di Stringhe**.

È esattamente il tipo di `argv` in `main(int argc, char **argv)` .

Come si immagina in memoria?

```
char **arr


  addr_0    addr_1    addr_2        array di puntatori (8 byte ciascuno)


  "ciao"   "mondo"   "!"           stringhe sparse nello Heap
```

Tema 11: Le `struct` e la Memoria Profonda (Deep Free)

> *[!IMPORTANT]*
> ***Fondamentale per l'esame e per il progetto di laboratorio!***

Il problema del Memory Leak

```
typedef struct {
    int id;
    char *nome;
} Utente;

Utente *u = malloc(sizeof(Utente));
u->nome = strdup("Mario Rossi");  // strdup fa un'ALTRA malloc!
```

Se fai solo `free(u);` **Memory Leak grave!** Hai distrutto il "contenitore" (la struct), ma la stringa `"Mario Rossi"` allocata nell'Heap è rimasta lì, **orfana e irraggiungibile**.

La regola: Deep Free (dal dentro verso il fuori)

```
free(u->nome);  // 1. Prima libero i campi interni dinamici
free(u);        // 2. Poi libero la struct stessa
```

Il trucco `sizeof(*a)`

```
// Invece di:
int *a = malloc(10 * sizeof(int));

// Scrivi:
int *a = malloc(10 * sizeof(*a));
```

> *[!TIP]*
> *`sizeof(*a)` significa "la dimensione dell'elemento puntato da `a`". Se cambi `int *a` in `long *a`, non devi modificare la `malloc`. È automatico.*

Ottimizzazione negli ordinamenti (`qsort`)

Se hai un array di `Utente` con campi enormi (es. matrici), ordinare l'array fisicamente richiede di copiare centinaia di byte a ogni scambio.

Best practice: Usa un **array di puntatori a struct** (`Utente **array`). Quando ordini, scambi solo i **puntatori** (8 byte), mentre le struct rimangono immobili nell'Heap. **Estremamente più veloce!**

Approccio	Costo per scambio
`Utente array[]`	Copia intera struct (centinaia di byte)
`Utente *array[]`	Copia solo puntatore (**8 byte**)

Tema 12: Lunghezza delle Stringhe (`strlen`)

```
#include <string.h>
size_t strlen(const char *s);
```

Parametro	Tipo	Descrizione
Ritorno	`size_t`	Numero di caratteri (escluso il terminatore `\0`).
`s`	`const char *`	La stringa da misurare.

Cosa fa: Scorre la stringa byte per byte e si ferma appena trova `\0`.

> *[!IMPORTANT]*
> ***Attenzione all'esame:*** *`strlen("ciao")` restituisce **4**. Ma per fare una `malloc` per copiarla, devi allocare `strlen("ciao") + 1` byte (il **+1** serve per il `\0`).*

Tema 13: Array di Struct vs Array di Puntatori — Trade-off

Riprendendo il Tema 11, ecco il ragionamento completo:

Approccio	Costo scambio	Accesso ai dati	Memoria extra
`Utente array[]`	Copia intera struct	Veloce (Cache Hit)	Nessuna
`Utente *array[]`	Copia solo 8 byte	Più lento (Cache Miss)	+8 byte/elemento

Il "prezzo da pagare" dei puntatori

1. **Spreco di spazio:** 8 byte in più per ogni puntatore.
2. **Cache Miss:** Il processore legge il puntatore, poi deve "saltare" in un punto lontano dell'Heap per trovare i dati reali.

> *[!TIP]*
> *Nonostante questi costi, per gli **ordinamenti** l'array di puntatori vince **quasi sempre**, perché il costo di copiare strutture enormi è molto superiore al costo dei cache miss.*

Tema 14: Spostarsi dentro i File (Accesso Random)

Finora abbiamo letto i file in modo sequenziale. Per leggere una posizione specifica si usano:

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
```

`fseek` — Muove il cursore

Parametro	Tipo	Descrizione
Ritorno	`int`	`0` successo, `-1` errore.
`stream`	`FILE *`	Il file aperto.
`offset`	`long`	Di quanti byte spostarsi (può essere negativo).
`whence`	`int`	Da dove partire a contare (vedi sotto).

Macro	Significato
`SEEK_SET`	Dall'**inizio** del file.
`SEEK_CUR`	Dalla **posizione corrente**.
`SEEK_END`	Dalla **fine** del file.

`ftell` — Dove mi trovo?

Restituisce la posizione esatta in byte rispetto all'inizio del file.

> *[!TIP]*
> *Trucco da esame — Calcolare la grandezza di un file:*
>
> ```
> fseek(file, 0, SEEK_END);      // Vai alla fine
> long dimensione = ftell(file);  // Leggi la posizione = dimensione in byte!
> fseek(file, 0, SEEK_SET);      // Torna all'inizio
> ```

Tema 15: Il Preprocessore e `#include`

Il processo di compilazione C è diviso in fasi. La prima è il **Preprocessore**. `#include` non "collega" una libreria: è un ordine testuale che dice *"Prendi tutto il contenuto del file header (`.h`) e incollalo qui"*.

Sintassi	Dove cerca il file
`#include <stdio.h>`	Nelle directory **standard di sistema** (es. `/usr/include`).
`#include "mio_file.h"`	**Prima** nella directory corrente, poi in quelle di sistema.

> *[!NOTE]*
> *Le parentesi angolari `< >` si usano per le librerie standard (POSIX, C). Le virgolette `" "` si usano per i file creati da te nel progetto.*

Tema 16: Puntatori a Funzione e `qsort`

In C, le funzioni risiedono in memoria (nella **Text Segment**). Hanno un indirizzo, quindi puoi creare un **puntatore a funzione**.

```
int (*f)(int, int);  // f può puntare a qualsiasi funzione (int, int)    int
```

> *[!IMPORTANT]*
> *Domanda d'esame: "Illustrare il significato del prototipo di `qsort` "*

`qsort` — `man 3 qsort`

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
```

Parametro	Tipo	Descrizione
`base`	`void *`	Indirizzo di partenza dell'array da ordinare.
`nmemb`	`size_t`	Numero di elementi nell'array.
`size`	`size_t`	Dimensione in byte di un singolo elemento (`sizeof(int)` , `sizeof(struct)` ...).

`compar`	`int (*)(const void *, const void *)`	**Puntatore alla tua funzione** di confronto.

Come funziona `compar` ?

`qsort` sa **spostare i byte** ma **non sa confrontare** i tuoi dati specifici. Chiede a te una funzione che:

- Prende **due elementi** (`const void *`)
- Restituisce: **negativo** (primo < secondo), **0** (uguali), **positivo** (primo > secondo)

> *[!NOTE]*
> *Questo è l'esempio perfetto di **"Polimorfismo" in C**: usa `void *` per rendere l'algoritmo compatibile con qualsiasi tipo di dato.*

Tema 17: Liste Concatenate (Linked List) vs Array

> *[!IMPORTANT]*
> *Teoria pura delle strutture dati, indispensabile per l'orale. Le liste sono anche argomento delle domande scritte su foglio.*

Confronto

Aspetto	Array	Linked List
Accesso	Casuale, istantaneo: `a[100]` **O(1)**	Sequenziale: devi scorrere da head **O(N)**
Dimensione	Fissa (serve `realloc` costosa)	Dinamica all'infinito (finché c'è RAM)
Inserimento in testa	Lentissimo: scalare N elementi **O(N)**	Istantaneo: stacca/riattacca puntatori **O(1)**
Memoria extra	Nessuna	+8 byte per campo `next` in ogni nodo

Struttura di un nodo

```
typedef struct Nodo {
    int dato;
    struct Nodo *next;  // Puntatore al nodo successivo
} Nodo;
```

17.1 Inserimento in Testa

```
// 1. Crei il nuovo nodo
Nodo *nuovo = malloc(sizeof(Nodo));
nuovo->dato = valore;

// 2. Il nuovo nodo punta al vecchio primo nodo
nuovo->next = head;

// 3. La testa della lista diventa il nuovo nodo
head = nuovo;
```

17.2 Distruzione della Lista (Trabocchetto d'Esame!)

Codice SBAGLIATO (da bocciatura)

```
while (head != NULL) {
    free(head);           // Nodo distrutto...
    head = head->next;    //    ERRORE! Leggi "next" da memoria già liberata!
}
```

Codice corretto — Iterativo (con variabile temporanea)

```
void distruggi_lista(Nodo *head) {
    Nodo *temp;
    while (head != NULL) {
        temp = head->next;  // 1. Salva dove andare dopo
        free(head);         // 2. Distruggi il nodo corrente
        head = temp;        // 3. Avanza usando l'indirizzo salvato
```

```
    }
}
```

Codice corretto — Ricorsivo (domanda frequente!)

```
void distruggi_lista_ricorsiva(Nodo *head) {
    if (head == NULL) {
        return;  // Caso base: lista vuota o finita
    }
    // Prima distruggi tutto il resto del "treno"...
    distruggi_lista_ricorsiva(head->next);

    // ...e solo quando i vagoni successivi sono distrutti, distruggi la testa
    free(head);
}
```

> *[!CAUTION]*
> ***La regola:*** *Devi* ***sempre*** *salvare* `head->next` ***prima*** *di fare* `free(head)` *. Nella versione ricorsiva, la chiamata ricorsiva usa* `head->next` *prima che* `free(head)` *venga eseguito, quindi funziona automaticamente.*

Tema 18: Portabilità dei Tipi — `ssize_t` e `size_t`

Perché non usiamo semplicemente `int` ?

Un `int` è bloccato a **4 byte** (32 bit, max ~2 miliardi). Se hai un file più grande di 2 GB e usi `int` per contarne i byte, il numero va in **overflow** (diventa negativo).

Tipo	Segno	Su 64-bit	Uso tipico
`size_t`	**Senza segno**	8 byte	Dimensioni, grandezze in byte (sempre positive).
`ssize_t`	**Con segno**	8 byte	Funzioni come `read` / `getline` : numero positivo O `-1` .

> *[!NOTE]*
> `size_t` *e* `ssize_t` *sono definiti dal sistema operativo per* ***scalare con l'architettura****.*

> *Su un PC a 64-bit occupano 8 byte, permettendo di gestire quantità immense.*

Tema 19: Leggere un File Riga per Riga — `getline`

> *[!IMPORTANT]*
> ***Domanda d'esame: "Come leggere le linee da un file di testo in C?"*** *Non si usa `fscanf`, si usa `getline`.*

```
#include <stdio.h>
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

Parametro	Tipo	Descrizione
Ritorno	`ssize_t`	Numero di caratteri letti (incluso `\n`). `-1` a EOF o errore.
`lineptr`	`char **`	Indirizzo del puntatore al buffer. Se `*lineptr` è `NULL`, fa `malloc`.
`n`	`size_t *`	Indirizzo della variabile con la capacità del buffer.
`stream`	`FILE *`	Il file da leggere.

La magia dell'allocazione (da spiegare all'orale)

```
char *buffer = NULL;
size_t n = 0;

while (getline(&buffer, &n, file) != -1) {
   printf("%s", buffer);
}
free(buffer);  // OBBLIGATORIO!
```

1. **Prima chiamata:** `buffer` è `NULL` `getline` fa `malloc` (es. 120 byte), aggiorna `buffer` e scrive `120` in `n`.
2. **Chiamate successive:** Riutilizza lo stesso buffer senza altre `malloc`.
3. **Riga lunghissima:** Se `n` non basta, fa internamente una `realloc` e aggiorna `n`.

I due valori a confronto

Valore	Significato
`n`	**Capacità fisica** del buffer in memoria (es. 120).
Valore di ritorno	**Caratteri effettivamente letti** (es. 40).

> *[!WARNING]*
> *Ricordati **sempre** di fare `free(buffer)` alla fine del programma, altrimenti hai un **memory leak**!*

Tema 20: Splittare le Stringhe — `strtok` e le Variabili Statiche

> *[!IMPORTANT]*
> ***Domanda d'esame: "Cosa vuol dire che `strtok` è MT-unsafe?"***

```
#include <string.h>
char *strtok(char *str, const char *delim);
```

Come funziona fisicamente in memoria (Il trucco del `\0`)

Con la stringa `"ciao;come;stai"` e delimitatore `";"` :

1. `strtok` cerca il primo `;`
2. **Sovrascrive** quel `;` con `\0`
3. Restituisce puntatore a `"ciao"`

> *[!CAUTION]*
> *La stringa originale è stata **"mutilata" in modo irreversibile**!*

I due cicli e il passaggio di `NULL`

```
char *token = strtok(mia_stringa, ";");  // Prima chiamata: passa la stringa
while (token != NULL) {
    printf("%s\n", token);
    token = strtok(NULL, ";");          // Successive: passa NULL!
}
```

Passare `NULL` dice: *"Continua a tagliare da dove ti eri fermata"*.

Variabili Statiche e MT-Unsafe

Il segreto: internamente `strtok` ha una variabile `static char *puntatore_interno`. Una variabile `static` dentro una funzione **non viene distrutta** nello Stack quando la funzione termina — sopravvive e mantiene il valore.

Scenario	Problema
Thread-1 fa `strtok` su stringa A	Entrambi modificano la **stessa** variabile statica interna
Thread-2 fa `strtok` su stringa B	I thread si rubano il cursore **crash o dati corrotti**

> *[!TIP]*
> *Per il multithreading si usa `strtok_r` (Reentrant), dove tu passi esplicitamente il puntatore di salvataggio come terzo parametro.*

Tema 21: La Compilazione Separata e i File Oggetto (`.o`)

> *[!IMPORTANT]*
> ***Domanda d'esame: "File oggetto e loro ruolo nella compilazione"***

Perché è SBAGLIATO fare `#include "funzioni.c"` ?

Problema	Spiegazione
Conflitto di simboli	Se includi un `.c` in due file, il Linker trova il codice duplicato errore `Multiple definition of function X`.
Lentezza estrema	Modifichi 1 file su 100 devi ricompilare **tutto** da zero.

La soluzione: Header (`.h`) + File Oggetto (`.o`)

Passo 1: Crea il file header

```
// funzioni.h — contiene SOLO i prototipi (dichiarazioni)
```

```
int somma(int a, int b);
```

Fai `#include "funzioni.h"` nei file `.c`. Dice al compilatore: *"Fidati, questa funzione esiste, la troverai dopo"*.

Passo 2: Compila separatamente con `-c`

```
gcc -c funzioni.c -o funzioni.o   # Crea il file oggetto
gcc -c main.c -o main.o           # Crea il file oggetto
```

I file `.o` contengono **codice macchina binario**, ma non sono eseguibili: sono frammentati (il `main.o` chiama `somma`, ma non sa a quale indirizzo si trovi).

Passo 3: Il Linker

```
gcc main.o funzioni.o -o programma_finale
```

Il Linker prende tutti i file oggetto, li **incastra come puzzle**, risolve i riferimenti incrociati e crea l'**eseguibile unico**.

Perché usiamo il Makefile?

> *[!TIP]*
> *Se modifici solo `funzioni.c`, il Makefile è intelligente: ricompila **solo** `funzioni.o` e rifà il Link finale. Non tocca `main.o` perché capisce che non è cambiato. Risparmio enorme di tempo!*

```
  funzioni.c          main.c

      gcc -c          gcc -c

  funzioni.o          main.o


            gcc (Linker)
```

```
        programma
```

Tema 22: Il Makefile

> *[!IMPORTANT]*
> ***Risponde alle due domande d'esame sul Makefile.***

Il Makefile è un file di testo letto dal programma `make` che automatizza la compilazione separata.

La Regola del Makefile (Target e Dipendenze)

```
target: dipendenza1.o dipendenza2.o
    comando da eseguire (es. gcc -o target ...)
```

Come ragiona `make` (Il trucco della Data)

`make` guarda il **Timestamp** (data di ultima modifica). Se la data di una dipendenza (es. `main.c`) è **più recente** del target (es. `main.o`), il codice è cambiato esegue il comando. Altrimenti *"Target is up to date"*.

> *[!NOTE]*
> *Se scrivi solo `make` nel terminale, eseguirà il **primo target** nel file (solitamente `all`).*

Le Variabili Automatiche

> *[!IMPORTANT]*
> ***Domanda frequente all'esame!***

Variabile	Significato
`$@`	Il nome del **Target**.
`$^`	I nomi di **TUTTE** le dipendenze, separate da spazi.
`$<`	Il nome della **PRIMA** dipendenza.

Esempio completo per l'esame

```
CC = gcc
CFLAGS = -Wall

programma: main.o lista.o
	$(CC) $(CFLAGS) -o $@ $^
# Equivale a: gcc -Wall -o programma main.o lista.o

main.o: main.c lista.h
	$(CC) $(CFLAGS) -c $<

lista.o: lista.c lista.h
	$(CC) $(CFLAGS) -c $<

clean:
	rm -f *.o programma
```

Il Target `clean`

Si inserisce sempre alla fine per fare pulizia. Non avendo dipendenze, si esegue chiamando esplicitamente `make clean`.

Tema 23: `qsort` — Il Problema del Casting dei Puntatori a Funzione

> *[!IMPORTANT]*
> *Questo dettaglio dimostra al prof che hai **davvero messo le mani nel codice**.*

Il problema

Se scrivi una funzione per confrontare due interi:

```
int confronta_interi(const int *a, const int *b) {
   return (*a - *b);
}
```

E poi chiami `qsort(array, n, sizeof(int), confronta_interi);` , GCC lancia un **Warning**. Perché? `qsort` esige `const void *` , tu passi `const int *` . Il C è molto rigido sui tipi dei puntatori a funzione.

Le due soluzioni (per prendere 30!)

Soluzione 1: Casting al volo (rapida ma "sporca")

```
qsort(array, n, sizeof(int), (__compar_fn_t) confronta_interi);
```

Soluzione 2: Casting interno (Standard, elegante)

```
int confronta_interi(const void *a, const void *b) {
    int valA = *((const int *) a);  // Cast da void* a int*, poi dereferenzio
    int valB = *((const int *) b);
    return (valA - valB);
}
```

> *[!TIP]*
> *La **Soluzione 2** è il modo corretto in C puro: la funzione accetta `void *` (compatibile con `qsort`) e il casting avviene internamente.*

Tema 24: Il Tranello di `getline` e il carattere `\n`

Se il file contiene la riga `Ciao` , fisicamente sul disco c'è `Ciao\n` . Quando `getline` la legge, **non toglie il "vai a capo"**:

```
Buffer: [ C | i | a | o | \n | \0 ]
```

Se fai `printf("%s\n", buffer);` , stamperai:

- `Ciao` + **a capo** (dal `\n` nel buffer) + **a capo** (dal `\n` della tua `printf`) = **doppio a capo!**

Trucco del mestiere

```
buffer[strcspn(buffer, "\n")] = 0;  // Rimuove brutalmente il \n
```

> *[!TIP]*
> *`strcspn(buffer, "\n")` restituisce l'indice del primo `\n` trovato. Sovrascrivendolo con `0` (`\0`), la stringa viene "accorciata" prima del newline.*

Tema 25: Ridirezioni dell'Output (Riepilogo Completo)

Comando	FD	Azione
`./prog > output.txt`	1	Ridirige **stdout** su file (**sovrascrive**).
`./prog >> output.txt`	1	Ridirige **stdout** su file (**append**, aggiunge in coda).
`./prog 2> errori.txt`	2	Ridirige **stderr** (`perror` , `fprintf(stderr,...)`) su file.
`./prog < input.txt`	0	Ridirige **stdin**: `scanf` / `getline` leggono da file, non da tastiera.
`./prog > out.txt 2> err.txt`	1+2	Separa output ed errori in due file distinti.

> *[!NOTE]*
> *I numeri 0, 1, 2 sono i **File Descriptor** standard: `0` = stdin, `1` = stdout, `2` = stderr.*

Tema 26: La Prova su Carta — Cancellazione "Deep" nelle Liste

> *[!IMPORTANT]*
> *Il prof ti darà una struct con un campo dinamico e ti chiederà di eliminare un nodo.*
> *Se fai solo `free` del nodo **senza liberare la stringa**, sei bocciato.*

Esempio da imparare (Ricorsivo)

```
typedef struct Nodo {
    char *nome;         // Allocato con strdup    malloc nascosta!
    struct Nodo *next;
} Nodo;

// Cancella TUTTI i nodi con un determinato nome
Nodo* cancella_nodo(Nodo *head, const char *target) {
```

```
    if (head == NULL) {
        return NULL;  // Caso base: lista finita
    }

    if (strcmp(head->nome, target) == 0) {
        Nodo *nodo_successivo = head->next;
        // DEEP FREE: prima il campo interno, poi la struct!
        free(head->nome);
        free(head);
        // Continua la ricorsione (in caso di duplicati)
        return cancella_nodo(nodo_successivo, target);
    } else {
        // Non è questo il nodo: tienilo e prosegui
        head->next = cancella_nodo(head->next, target);
        return head;
    }
}
```

> *[!CAUTION]*
> ***Ordine obbligatorio:*** *`free(head->nome)` **prima** di `free(head)`. Se fai il contrario, perdi l'accesso al campo `nome` e hai un memory leak.*

Tema 27: Alberi Binari di Ricerca (ABR) vs Liste

Confronto di performance

Struttura	Ricerca	Complessità	Esempio (1M elementi)
Lista (non ord.)	Sequenziale	**O(N)**	Fino a 1.000.000 controlli
ABR (bilanciato)	Dimezzamento	**O(log N)**	Massimo **~20** controlli

Come funziona l'ABR

Partendo dalla radice, tutti i nodi a **sinistra** sono più piccoli, tutti quelli a **destra** sono più grandi. A ogni passo **dimezzi** gli elementi da controllare.

Stampa condizionale con puntatore a funzione

```
// Visita In-Order: stampa i nodi in ordine crescente, solo se la condizione è vera
void stampa_se(NodoABR *radice, int (*condizione)(NodoABR *)) {
    if (radice == NULL) return;

    stampa_se(radice->sinistra, condizione);

    if (condizione(radice) == 1) {  // Chiamo la funzione puntatore!
        printf("%d\n", radice->valore);
    }

    stampa_se(radice->destra, condizione);
}
```

> *[!TIP]*
> *Invece di scrivere 10 funzioni diverse (`stampa_pari`, `stampa_positivi` ...), ne scrivi **una sola generica** che accetta un puntatore a funzione "condizione". Stesso principio di `qsort`.*

Tema 28: Funzioni "Spia" che Allocano Dinamicamente (Riepilogo)

> *[!IMPORTANT]*
> ***Regola d'oro del C:** "Se la libreria fa una `malloc` per te, la responsabilità della `free` diventa tua."*

Funzione	Cosa fa di nascosto	free necessaria?
`getline(&buf, &n, file)`	Prima volta: `malloc`. Successive: riusa. Se serve: `realloc`.	`free(buf)`
`strdup("ciao")`	`strlen` + `malloc` + `strcpy` in un colpo solo.	`free(ptr)`
`asprintf(&str, "Nome: %s", n)`	Calcola lunghezza, `malloc` esatta, scrive, aggiorna puntatore.	`free(str)`
`fscanf(file, "%ms", &str)`	**Solo con `%ms`!** `fscanf` normale NON alloca.	`free(str)`

> *[!WARNING]*
> *`fscanf` normale **non alloca dinamicamente**! Lo fa **solo** con il modificatore POSIX speciale `%ms` e passando l'indirizzo del puntatore.*

Tema 29: Thread-Safety — `strerror` vs `strerror_r`

Chiusura perfetta dopo aver parlato di `strtok` (MT-Unsafe).

`strerror` — MT-Unsafe

```
#include <string.h>
char *strerror(int errnum);
```

Come fa a restituire un `char *` senza chiederti la `free`? Restituisce un puntatore a un **buffer statico condiviso**. Se due Thread chiamano `strerror` contemporaneamente per errori diversi uno **sovrascrive** il buffer dell'altro.

`strerror_r` — MT-Safe (Reentrant)

```
int strerror_r(int errnum, char *buf, size_t buflen);
```

Parametro	Tipo	Descrizione
Ritorno	`int`	`0` successo, numero errore in caso di fallimento.
`errnum`	`int`	Il codice errore (es. `errno`).
`buf`	`char *`	**Il tuo buffer** dove la funzione scrive il messaggio.
`buflen`	`size_t`	Dimensione del buffer.

```
char mio_errore[256];
strerror_r(errno, mio_errore, sizeof(mio_errore));
// mio_errore è locale al thread nessuna collisione!
```

> *[!NOTE]*
> *Il suffisso `_r` in POSIX significa **Reentrant** (Thread-Safe). La funzione non usa*

> *variabili statiche interne: sei tu a fornire il buffer (locale allo Stack del Thread), eliminando ogni rischio di collisione.*

Tema 30: `void *` — Promozione Implicita e Dettagli su `qsort`

Perché `int *a = malloc(...)` funziona senza cast?

In C, `void *` viene **promosso implicitamente** a qualsiasi altro tipo di puntatore. Il compilatore capisce da solo la conversione.

> *[!NOTE]*
> *In **C++** questo darebbe errore e richiederebbe un cast esplicito `(int *)`. In C puro è legale e prassi comune.*

Perché `qsort` ha bisogno di `nmemb` e `size`?

Con un `int *`, fare `ptr + 1` salta avanti di 4 byte (il compilatore sa la dimensione). Ma `qsort` riceve un `void *` il compilatore **non sa** quanto è grande un `void`.

Per spostarsi nell'array, `qsort` fa la matematica a mano:

```
indirizzo_base + (indice × size)
```

Ecco perché devi passargli la dimensione esatta con `sizeof`.

Perché `const void *` nella funzione di confronto?

`const` è una **promessa**: *"Ti passo i puntatori per guardare questi elementi, ma il lucchetto `const` ti impedisce di modificarli, rovinandomi l'array"*.

Tema 31: La Doppia Vita di `static`

> *[!IMPORTANT]*
> ***Domanda d'esame frequentissima!*** *`static` fa due cose completamente diverse.*

A. Variabile Statica (dentro una funzione) — Cambia il Ciclo di Vita

```
void conta() {
    static int n = 0;  // Inizializzata solo la PRIMA volta!
    n++;
    printf("%d\n", n);
}
```

Senza `static`	Con `static`
Nasce nello **Stack**, muore al `return`	Nasce nel **Data Segment**, vive per sempre
Valore perso tra le chiamate	**Mantiene il valore** tra le chiamate

> *[!NOTE]*
> *È il meccanismo usato da `strtok` per ricordare il cursore tra chiamate successive.*

B. Funzione/Variabile Globale Statica — Cambia la Visibilità

```
static void calcola() { ... }  // Visibile SOLO in questo file .c
```

Senza `static`	Con `static`
Funzione **pubblica**: altri file possono chiamarla	Funzione **privata**: invisibile fuori dal file
Rischio errore Linker (`Multiple definition`)	Nessun conflitto anche con nomi identici

---

Tema 32: File Testuali vs File Binari (`fwrite` / `fread`)

Confronto

Aspetto	**Testuale (`fprintf` / `fscanf`)**	**Binario (`fwrite` / `fread`)**
Formato	Caratteri ASCII leggibili	Dump esatto della RAM
Intero 1234567	**7 byte** (1 per cifra)	**4 byte** (sizeof(int), sempre uguale)
Velocità	Lenta (conversione necessaria)	**Velocissima** (copia diretta)

Leggibilità	Apribile con editor di testo	Illeggibile senza strumenti appositi

Prototipi e il `const`

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Funzione	Puntatore	Perché?
`fwrite`	`const void *`	**Legge** la tua RAM non la modifica. `const` rassicura.
`fread`	`void *`	**Scrive** nella tua RAM non può essere `const`.

Esempio da esame

```
int numero;
fread(&numero, sizeof(int), 1, file);  // Legge 4 byte dal file dentro 'numero'
```

Tema 33: `rewind` e il Comando `od` (Octal Dump)

`rewind` — Scorciatoia

```
void rewind(FILE *stream);
// Equivale esattamente a: fseek(stream, 0, SEEK_SET);
```

Riporta il cursore al byte 0 (inizio del file).

`od` — Leggere file binari da terminale

Se apri un file binario con `cat`, vedrai simboli illeggibili. Per leggere il contenuto reale:

Comando	Output
`od -c mio_file.bin`	Byte interpretati come **caratteri** (`\n`, `\0`...)
`od -x mio_file.bin`	Byte in formato **esadecimale** (debug di basso livello)

Parte II — Programmazione di Sistema

Tema 34: System Call vs Library Functions (L'I/O in C)

Le funzioni di libreria C (**sezione 3** del `man`) offrono comodità e ottimizzazione (buffering), ma sotto il cofano devono sempre chiamare le **System Call POSIX** (**sezione 2** del `man`) per parlare con il Kernel.

34.1 Apertura File: `open` vs `fopen`

System Call: `open` — `man 2 open`

```
int open(const char *pathname, int flags, mode_t mode);
```

Parametro	Tipo	Descrizione
Ritorno	`int`	File Descriptor (`fd`): intero non negativo, indice nella tabella dei file aperti del kernel. `-1` in errore.
`pathname`	`const char *`	Percorso del file.
`flags`	`int`	Operazioni bit-a-bit (OR logico `|`) per la modalità di apertura (es. `O_RDONLY`, `O_WRONLY`, `O_CREAT`).
`mode`	`mode_t`	*(Opzionale, solo con `O_CREAT`)* Permessi di creazione in ottale (es. `0644`).

Library Function: `fopen` — `man 3 fopen`

```
FILE *fopen(const char *pathname, const char *mode);
```

Parametro	Tipo	Descrizione
Ritorno	`FILE *`	Puntatore a `struct FILE`: contiene il fd + un **buffer in user-space** + contatori. `NULL` in caso di errore.

`pathname`	`const char *`	Percorso del file.
`mode`	`const char *`	Stringa intuitiva: `"r"` (lettura), `"w"` (scrittura, crea/tronca), `"a"` (append). Con `"w"` applica `O_CREAT` di default.

34.2 Lettura File: `read` vs `fread`

> *[!IMPORTANT]*
> ***Domanda d'esame frequente!*** *La differenza tra `read` e `fread` è fondamentale.*

System Call: `read` — `man 2 read`

```
ssize_t read(int fd, void *buf, size_t count);
```

Parametro	Tipo	Descrizione
Ritorno	`ssize_t`	Intero con segno: numero di byte letti, `0` = EOF, `-1` = errore.
`fd`	`int`	File Descriptor da cui leggere.
`buf`	`void *`	Puntatore alla memoria dove il kernel scriverà i byte (array o `malloc`).
`count`	`size_t`	Numero totale di byte da leggere.

> *[!WARNING]*
> ***Non è bufferizzata.*** *Ogni chiamata a `read` fa un **context switch** (user-space kernel-space), che è costoso.*

Library Function: `fread` — `man 3 fread`

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Parametro	Tipo	Descrizione
Ritorno	`size_t`	Intero senza segno: numero di **elementi** letti (non byte!). Se < `nmemb` errore o EOF.

`ptr`	`void *`	Buffer di destinazione.
`size`	`size_t`	Dimensione in byte di un singolo elemento (es. `sizeof(int)`).
`nmemb`	`size_t`	Numero di elementi da leggere. Totale: `size × nmemb` byte.
`stream`	`FILE *`	Puntatore a `FILE`.

> *[!TIP]*
> ***È bufferizzata.*** *`fread` legge pezzi grandi dal disco (chiamando internamente `read`) e li salva nel buffer della `struct FILE`. Le letture successive prendono i dati dal buffer, evitando chiamate continue al kernel. Usare `ferror()` o `feof()` per distinguere errore da EOF.*

Tema 35: Permessi e UMASK

> *[!IMPORTANT]*
> ***Domanda d'esame frequente!***

La `open` richiede i permessi (`mode`) in **formato ottale**. In C, un numero ottale inizia con `0`.

Esempio: `0777` in binario `111 111 111` `rwx` per Utente, Gruppo e Altri.

Cos'è la UMASK?

La **umask** (*User file-creation mode mask*) è una maschera di bit impostata dalla shell che determina quali permessi devono essere **rimossi (negati)** quando un processo crea un nuovo file. Serve per **sicurezza**: evita che i programmi creino file eseguibili o scrivibili da tutti senza volerlo.

Formula

```
Permessi effettivi = mode & (~umask)
```

> *mode AND (NOT umask)*

Esempio pratico

Elemento	Ottale	Binario
`mode` richiesto	`0666`	`110 110 110`
`umask`	`0022`	`000 010 010`
`~umask`		`111 101 101`
Risultato finale	**`0644`**	**`110 100 100`**

> *[!NOTE]*
> *Con `mode = 0666` e `umask = 0022`, i permessi reali saranno `0644`: lettura/scrittura per il proprietario, solo lettura per gruppo e altri.*

Tema 36: Memoria Dinamica (`sbrk`)

Le funzioni `malloc` e `free` sono **funzioni di libreria**. Sono complessi "gestori di memoria" che tengono traccia di quali blocchi di RAM sono liberi e quali occupati all'interno dell'**Heap**.

Quando `malloc` finisce la memoria disponibile nel suo "pool" in user-space, chiama la **System Call** `sbrk`:

```
void *sbrk(intptr_t increment);
```

Parametro	Tipo	Descrizione
Ritorno	`void *`	Il vecchio indirizzo del program break.
`increment`	`intptr_t`	Numero di byte di cui spostare il program break (limite superiore dell'Heap).

> *[!NOTE]*
> *`sbrk` è "bruttissima" da usare direttamente: si limita a spostare più in alto il program break. Usiamo `malloc` proprio perché maschera questa complessità.*

Tema 37: Creazione e Controllo dei Processi

Comandi utili nella shell

Comando	Descrizione
`ps`	Mostra i processi attivi
`echo $$`	Stampa il PID della shell corrente
`comando &`	Avvia un processo in background
`kill -9 PID`	Manda il segnale `SIGKILL` (non bloccabile) al processo

37.1 Creare un processo: `fork`

> *[!IMPORTANT]*
> ***Domanda d'esame: "Parlare della funzione fork"***

```
#include <unistd.h>
pid_t fork(void);
```

Cosa fa: Crea un processo figlio che è un **clone esatto** del padre (stesso codice, stessa memoria, stessi file aperti). Il kernel Linux usa l'ottimizzazione **Copy-on-Write** (non copia fisicamente la RAM, ma la condivide finché uno dei due non ci scrive).

Valori di ritorno (`pid_t`)

> *"Chiamata una volta, ritorna due volte"*

Contesto	Valore restituito
Al Padre	PID del figlio (intero positivo)
Al Figlio	`0` — per conoscere il suo vero PID deve usare `getpid()`
Errore	`-1` al padre, nessun figlio creato

Funzioni ausiliarie

```
pid_t getpid(void);   // Ritorna il PID del processo corrente
pid_t getppid(void);  // Ritorna il PID del PADRE (Parent PID)
```

37.2 Terminare un processo: `exit`

```
void exit(int status);
```

Termina il processo. Effettua le seguenti operazioni:

1. Chiude i file descriptor
2. Svuota i buffer I/O (`fflush`)
3. Passa il valore `status` al processo padre

> *[!WARNING]*
> *Il processo è "morto", ma le sue informazioni (come il codice di ritorno `status`)* ***rimangono salvate nel kernel*** *finché il padre non le legge con `wait()` .*

37.3 Aspettare un processo: `wait` e i Processi Zombie

> *[!IMPORTANT]*
> ***Domanda d'esame frequente!***

```
#include <sys/wait.h>
pid_t wait(int *wstatus);
```

Parametro	Tipo	Descrizione
Ritorno	`pid_t`	PID del figlio terminato, oppure `-1` se non ci sono figli.
`wstatus`	`int *`	Puntatore a intero dove il kernel scrive lo status del figlio. Si analizza con macro come `WIFEXITED(wstatus)` .

Cosa fa: Blocca l'esecuzione del padre finché **uno qualsiasi** dei figli non termina.

Cosa sono i Processi Zombie?

Se un figlio chiama `exit()` ma il padre **non ha ancora chiamato `wait()`** , il figlio diventa uno **Zombie** (visibile con `ps` , marcato con `Z`).

Aspetto	Dettaglio
Memoria codice/variabili	Non occupata (liberata)
Slot nella Tabella Processi	**Ancora occupato** — mantiene lo status di ritorno per il padre

Rischio	Migliaia di zombie esaurimento PID blocco del sistema
Soluzione	`wait()` fa la "**mietitura**" (*reaping*) dello zombie e libera lo slot

> *[!CAUTION]*
> *Se un programma genera migliaia di zombie senza mai fare* `wait()` *, il sistema* ***esaurisce i PID disponibili*** *e diventa impossibile creare nuovi processi!*

Tema 38: Gestione Errori e Debug (Le Basi)

`errno` e `perror`

Quando una System Call o una funzione di libreria fallisce, restituisce solitamente `-1` (o `NULL`) e imposta una **variabile globale intera** chiamata `errno` con il codice specifico dell'errore (es. `EACCES` per permesso negato).

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

perror("messaggio");  // Legge errno e stampa l'errore in formato testuale
```

Macro `__FILE__` e `__LINE__`

Sono **macro del preprocessore C**. Sostituiscono rispettivamente il nome del file sorgente e il numero di riga corrente.

> *[!TIP]*
> ***Perché si usano?*** *Se hai 4* `fork` *nel codice e una fallisce, fare* `perror("Errore fork")` *non ti dice quale ha fallito. Invece:*
>
> ```
> printf("Errore nel file %s alla riga %d\n", __FILE__, __LINE__);
> ```
>
> *Ti fa trovare subito il bug.*

Tema 39: Attesa dei Processi e `exec`

39.1 `waitpid` — Attesa selettiva

Abbiamo visto `wait`, ma per controlli più fini si usa `waitpid`:

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Parametro	Tipo	Descrizione
Ritorno	`pid_t`	PID del figlio terminato, `0` se `WNOHANG` e nessun figlio terminato, `-1` in errore.
`pid`	`pid_t`	`-1` = qualsiasi figlio (come `wait`). Oppure il PID di un figlio specifico.
`wstatus`	`int *`	Puntatore all'intero dove verrà salvato lo stato di uscita.
`options`	`int`	Es. `WNOHANG`: non bloccante. Se il figlio non è terminato, restituisce `0` senza bloccarsi.

Macro per leggere `wstatus`

Non puoi leggere l'intero `wstatus` direttamente, perché il kernel ci impacchetta diverse informazioni. Devi usare le macro:

Macro	Descrizione
`WIFEXITED(status)`	Restituisce vero (non zero) se il figlio è terminato "normalmente" (`exit()` o fine `main`).
`WEXITSTATUS(status)`	Estrae solo gli **8 bit meno significativi** del valore passato alla `exit()`.

> *[!IMPORTANT]*
> ***Domanda d'esame:*** *`WEXITSTATUS(status)` va usata **SOLO SE** `WIFEXITED(status)` è vero. Se il figlio ha fatto `exit(5)`, questa macro restituisce `5`.*

39.2 Famiglia `exec` (es. `execl`)

Dopo una `fork`, il figlio è un clone del padre. Spesso si usa una funzione `exec` per **sostituire completamente l'immagine in memoria** del figlio con un nuovo

programma.

```
int execl(const char *pathname, const char *arg, ... /* (char *) NULL */);
int execvp(const char *file, char *const argv[]);
```

> *[!NOTE]*
> *Il **PID rimane lo stesso**, ma il codice diventa quello del nuovo programma (es.* `/bin/ls` *). Se* `exec` *ha successo, **non ritorna mai** — il vecchio codice non esiste più. Ritorna solo in caso di errore (* `-1` *).*

Tema 40: IPC — Le Pipe (Con Nome e Senza Nome)

> *[!IMPORTANT]*
> *Domanda d'esame frequente!*

Le pipe sono **canali di comunicazione unidirezionali** gestiti dal kernel (in RAM).

40.1 Pipe Senza Nome (`pipe`)

Usate **solo tra processi imparentati** (padre-figlio).

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

Parametro	Tipo	Descrizione
Ritorno	`int`	`0` in caso di successo, `-1` in errore.
`pipefd[0]`	`int`	Lato di **LETTURA** *(ricorda: lo Standard Input è 0)*.
`pipefd[1]`	`int`	Lato di **SCRITTURA** *(ricorda: lo Standard Output è 1)*.

Comportamento bloccante e sincronizzazione

Situazione	Comportamento
`read` su pipe vuota	**Si blocca** finché non arriva un dato.
`write` su pipe piena	**Si blocca** finché non si libera spazio nel buffer.

Scritture piccole	Sono **atomiche** (garantite indivisibili).

> *[!CAUTION]*
> ***Regola d'oro per la terminazione:*** *La* `read` *restituirà* `0` *(EOF)* ***SOLO SE*** *tutti i processi hanno chiuso il lato di scrittura* `pipefd[1]` *con* `close()` *. Se dimentichi di chiudere una pipe, il programma va in* ***stallo (deadlock)****.*

40.2 Pipe Con Nome (Named Pipes / FIFO)

Rispondono alla domanda: *"Come comunicano due processi che NON hanno fatto fork (non imparentati)?"*

Creano un **"file speciale"** visibile nel File System. Chiunque conosca il percorso del file può scriverci o leggerci.

`mkfifo` — Differenza tra `man 1` e `man 3`

> *[!IMPORTANT]*
> ***Domanda d'esame: mkfifo (1) e (3)***

Sezione del man	Cosa è	Esempio
`man 1 mkfifo`	**Comando da terminale** (shell)	`mkfifo mia_pipe`
`man 3 mkfifo`	**Funzione in C** (libreria) da usare nel codice	Vedi prototipo sotto

```
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

Parametro	Tipo	Descrizione
Ritorno	`int`	`0` successo, `-1` errore.
`pathname`	`const char *`	Percorso del file speciale FIFO da creare.
`mode`	`mode_t`	Permessi di creazione (soggetti alla umask).

> *[!NOTE]*
> ***Sincronizzazione della*** `open` ***:*** *Se un processo chiama* `open("mia_pipe", O_RDONLY)` *, si* ***blocca*** *finché un altro processo non chiama* `open("mia_pipe", O_WRONLY)` *, e viceversa. Devono incontrarsi "all'appuntamento".*

Tema 41: La Memoria Condivisa (Shared Memory / `mmap`)

> *[!IMPORTANT]*
> *Domanda d'esame!*

Le pipe sono lente perché bisogna fare `read` / `write` e copiare i dati dal Kernel allo User Space. La **Memoria Condivisa** risolve il problema mappando una zona di RAM direttamente nello spazio di indirizzamento di due processi.

Il workflow in 4 passi

Passo 1: `shm_open` — Crea/Apre l'oggetto condiviso

```
#include <sys/mman.h>
#include <fcntl.h>
int shm_open(const char *name, int oflag, mode_t mode);
```

Parametro	Tipo	Descrizione
Ritorno	`int`	File Descriptor. Il file fisicamente **non va su disco**, ma in RAM (`/dev/shm`).
`name`	`const char *`	Nome dell'oggetto condiviso (es. `"/mia_shm"`).
`oflag`	`int`	Flag di apertura (es. `O_CREAT
`mode`	`mode_t`	Permessi (es. `0666`).

Passo 2: `ftruncate` — Imposta la dimensione

```
int ftruncate(int fd, off_t length);
```

Parametro	Tipo	Descrizione
Ritorno	`int`	`0` successo, `-1` errore.
`fd`	`int`	File Descriptor dell'oggetto condiviso.

`length`	`off_t`	Dimensione desiderata (es. `sizeof(int) * 100`). Appena creato ha 0 byte.

Passo 3: `mmap` — Mappa il file in memoria (La vera magia)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Parametro	Tipo	Descrizione
Ritorno	`void *`	Puntatore usabile come un **normale array in C**. `MAP_FAILED` in errore.
`addr`	`void *`	Di solito `NULL` (lasci scegliere al kernel).
`length`	`size_t`	Dimensione della mappatura in byte.
`prot`	`int`	Protezione: `PROT_READ`, `PROT_WRITE`, combinabili con `|`.
`flags`	`int`	Es. `MAP_SHARED` (modifiche visibili agli altri processi).
`fd`	`int`	File Descriptor dell'oggetto condiviso.
`offset`	`off_t`	Offset nel file da cui iniziare la mappatura (di solito `0`).

> *[!TIP]*
> *Tutto ciò che scrivi tramite il puntatore restituito, l'altro processo lo vede **all'istante**, senza chiamare `write()` !*

Passo 4: Chiusura e Distruzione

```
int munmap(void *addr, size_t length);   // Scollega la memoria dal processo (come una
int shm_unlink(const char *name);        // Rimuove il "nome" da /dev/shm
```

> *[!IMPORTANT]*
> ***Domanda frequente — `shm_unlink` :** Rimuove il "nome" da `/dev/shm`, ma la **memoria effettiva in RAM** verrà distrutta dal sistema operativo **solo quando tutti i processi** che l'avevano mappata avranno fatto la `munmap`.*

Tema 42: Semafori Con Nome (POSIX)

> *[!IMPORTANT]*
> ***Domanda d'esame frequente!***

Il problema della memoria condivisa è che se due processi scrivono sulla stessa variabile contemporaneamente (**Race Condition**), i dati si corrompono. Servono i **Semafori**.

I semafori con nome si trovano in `/dev/shm` e possono essere aperti da **processi non imparentati**.

42.1 `sem_open` — Creazione

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

Parametro	Tipo	Descrizione
Ritorno	`sem_t *`	Puntatore al semaforo (**non un intero!**). `SEM_FAILED` in errore.
`name`	`const char *`	Nome del semaforo (es. `"/mio_sem"`).
`oflag`	`int`	Flag (es. `O_CREAT
`mode`	`mode_t`	Permessi (es. `0666`).
`value`	`unsigned int`	Valore iniziale (es. `1` se usato come **Mutex**).

42.2 `sem_wait` (P) e `sem_post` (V)

```
int sem_wait(sem_t *sem);  // Decrementa. Se è 0, si BLOCCA.
int sem_post(sem_t *sem);  // Incrementa. Se c'erano processi bloccati, ne sveglia uno.
```

Operazione	Nome classico	Effetto

`sem_wait`	**P** (Proberen)	Decrementa il contatore. Se il valore è `0`, il processo si **blocca**.
`sem_post`	**V** (Verhogen)	Incrementa il contatore. Se c'erano processi bloccati, ne **sveglia uno**.

> *[!NOTE]*
> ***Domanda — `sem_post` :*** *È un'operazione **atomica** fornita dal kernel. Garantisce che l'incremento del valore interno del semaforo non venga interrotto da altri thread o processi.*

42.3 `sem_unlink` — Distruzione

> *[!IMPORTANT]*
> ***Domanda d'esame specifica!***

```
int sem_unlink(const char *name);
```

Funziona come `shm_unlink`:

1. **Rimuove immediatamente** il file dal file system virtuale (`/dev/shm`), impedendo che nuovi processi possano fare `sem_open` su quel nome.
2. **Tuttavia**, la struttura dati nel kernel **non viene deallocata** finché tutti i processi che hanno già il semaforo aperto non chiamano `sem_close()` .

Tema 43: I Segnali — Interruzioni Software Asincrone (`man 7 signal`)

I segnali sono l'unico modo che il Kernel (o un altro processo) ha per dire a un programma: *"Fermati immediatamente e gestisci questo evento"*.

Perché usiamo i nomi (`SIGINT`) e non i numeri (`2`)?

La corrispondenza Nome Numero **varia in base all'architettura hardware** (x86, ARM...). Scrivendo `SIGINT` il compilatore inserisce il numero giusto per la macchina specifica.

Segnali principali (da sapere a memoria)

Segnale	Generato da	Azione Default	Note
`SIGINT`	`Ctrl+C`	Termina il processo	Interrupt
`SIGQUIT`	`Ctrl+\`	Termina + **Core Dump**	Salva la RAM per debug
`SIGSEGV`	Accesso memoria invalida	Termina + Core Dump	Segmentation Fault
`SIGKILL`	`kill -9`	Termina (brutale)	**NON catturabile/ignorabile**
`SIGSTOP`	`Ctrl+Z`	Congela il processo	**NON catturabile/ignorabile**
`SIGCONT`	`fg` / `kill -CONT`	Riprende un processo fermo	

> *[!IMPORTANT]*
> ***Domanda d'esame:*** *`SIGKILL` e `SIGSTOP` sono gli unici due segnali "divini". Il programmatore **NON può** bloccarli, ignorarli o catturarli.*

Come si inviano i segnali

```
#include <signal.h>
int kill(pid_t pid, int sig);             // A un PROCESSO (o gruppo)
int pthread_kill(pthread_t thread, int sig); // A un SINGOLO THREAD
```

> *[!NOTE]*
> *`kill` non significa per forza "uccidi". `kill(pid, SIGCONT)` **sveglia** un processo. Il nome è storico.*

Tema 44: Architettura Kernel dei Segnali (Le 3 Strutture Dati)

> *[!IMPORTANT]*
> ***Cuore dell'orale!*** *Il Kernel gestisce 3 strutture dati per i segnali.*

A. Signal Handler Array (UNA per PROCESSO)

Tabella che associa ogni segnale all'azione da compiere. **Tutti i thread condividono gli stessi handler.**

Azione possibile	Significato
`SIG_DFL`	Azione **default** (di solito: termina).
`SIG_IGN`	**Ignora** il segnale.
`&mia_funzione`	Esegui la tua funzione **custom** (Signal Handler).

B. Pending Signal Bitmap (UNA per THREAD)

Mappa di bit (64 bit). Se `SIGINT` (n° 2) arriva ma il thread lo blocca il Kernel setta il **2° bit** a `1`. Il segnale è **Pendente**.

> *[!CAUTION]*
> ***Trabocchetto d'esame:*** *Se arrivano* ***5 SIGINT*** *mentre il thread li blocca, il bit viene settato a* `1` *solo la* ***prima volta****. Quando il thread sblocca, l'handler viene eseguito* ***UNA SOLA VOLTA****. I segnali POSIX standard* ***non si accodano****.*

C. Signal Mask (UNA per THREAD)

Mappa di bit che dice: *"Non voglio essere disturbato dai segnali con* `1` *in questa maschera"*. Se mascherato, il segnale resta Pending.

Il Ciclo di Vita di un segnale

1. Il segnale arriva il Kernel lo segna nella **Pending Bitmap**
2. Al prossimo **Context Switch** (ritorno da syscall/interrupt), il Kernel controlla:
    - Se il segnale è **mascherato** resta Pending
    - Se **non** è mascherato il Kernel crea un **frame fittizio** sullo Stack
3. Il **Program Counter** viene spostato alla funzione handler
4. Quando l'handler fa `return` syscall nascosta `sigreturn` ripristina i registri il `main` riprende come se nulla fosse

> *[!WARNING]*
> ***System Call lente:*** *Se il segnale arriva durante una* `read()` *bloccante su una pipe vuota, la* `read` *si interrompe, restituisce* `-1` *e imposta* `errno = EINTR` *(Interrupted system call). Il programmatore deve gestirlo!*

Multithreading e Segnali

Tipo di segnale	Destinazione
Sincrono (errore)	Va al **thread che ha causato l'errore** (es. `SIGFPE` per divisione/0).
Asincrono (esterno)	Il Kernel sceglie **a caso** un thread la cui mask non lo blocca.

> *[!TIP]*
> *Questa casualità è terribile per la programmazione! Per questo si crea il **"Thread Gestore dei Segnali"** (trattato nel prossimo blocco).*

Tema 45: Programmare i Segnali — `sigaction`

> *[!WARNING]*
> ***Non usare mai*** *`signal()` **!** Ha comportamenti imprevedibili. Si usa* *`sigaction`* *(* *`man 2 sigaction`* *).*

Il prototipo del handler

```
void mio_handler(int signum) {
   // signum contiene il numero del segnale ricevuto
   // ATTENZIONE: qui puoi usare solo funzioni async-signal-safe!
}
```

Configurare con `sigaction`

```
struct sigaction sa;
sa.sa_handler = mio_handler;     // Puntatore alla tua funzione
sigemptyset(&sa.sa_mask);        // Inizializza maschera vuota
sigaddset(&sa.sa_mask, SIGTERM); // Blocca SIGTERM durante l'handler
sa.sa_flags = 0;

sigaction(SIGINT, &sa, NULL);    // Applica le regole a SIGINT
```

Spiegazione per l'esame

Con questo codice dici al SO:

1. *"Quando arriva* `SIGINT` *, esegui* `mio_handler` *"*
2. *"Mentre esegui l'handler, blocca* `SIGINT` *"* (automatico)
3. *"Blocca anche* `SIGTERM` *"* (tramite `sa_mask`)
4. *"Appena l'handler finisce, sblocca tutto"*

> *[!IMPORTANT]*
> ***Perché bloccare segnali durante l'handler?*** *Se stai chiudendo file in risposta a* `SIGINT` *e l'utente preme di nuovo* `Ctrl+C` *, non vuoi che l'handler venga interrotto da un altro* `SIGINT` *(ricorsione distruttiva).*

Tema 46: Dettagli su `sigaction` e Segnali Custom

Il terzo parametro di `sigaction` (`oldact`)

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

Se passi un puntatore valido (invece di `NULL`), il Kernel **salva in `oldact` le regole precedenti** prima di impostare le nuove. Utile per modificare temporaneamente un handler e poi **ripristinare** il vecchio comportamento.

Interazione con Variabili Globali

L'handler riceve **solo** `int signum` non puoi passargli puntatori o dati del main. L'unico modo per comunicare è usare **variabili globali**, ma:

```
volatile int flag_uscita = 0;  // OBBLIGATORIO: volatile!
```

> *[!CAUTION]*
> *Senza* `volatile` *, il compilatore potrebbe* ***ottimizzare*** *la variabile in un registro CPU. L'handler la modifica in memoria, ma il* `main` *legge il registro vecchio e* ***non se ne accorge****.*

Segnali Custom: `SIGUSR1` e `SIGUSR2`

Il Kernel fornisce **due segnali vuoti**, senza significato predefinito, lasciati al programmatore per far comunicare processi:

```
kill(pid_B, SIGUSR1);  // Processo A dice a B: "Ho finito, tocca a te"
```

Tema 47: Multi-Thread Safe (MT-Safe)

> *[!IMPORTANT]*
> *Domanda d'esame chiave!*

Definizione: Una funzione è MT-Safe se può essere chiamata **contemporaneamente** da più Thread, producendo sempre risultati corretti indipendentemente dall'**interleaving** (ordine di esecuzione).

I nemici del MT-Safe

Nemico	Perché è pericoloso
Variabili Globali	Condivise da tutti i thread. Race condition in lettura/scrittura.
Variabili Statiche	Esistono in copia unica (es. cursore di `strtok`). Stessa race condition.

La Soluzione: I Mutex

```
pthread_mutex_lock(&lock);
// --- Sezione Critica: tocca variabili condivise ---
contatore++;
pthread_mutex_unlock(&lock);
```

Tema 48: Async-Signal-Safe (Domanda da Lode!)

> *[!IMPORTANT]*
> *Riferimento:* `man 7 signal-safety`

Definizione: Una funzione è Async-Signal-Safe se funziona correttamente anche se viene **interrotta da un segnale** e l'handler chiama **la stessa funzione** prima che la prima invocazione finisca.

Perché i Mutex NON funzionano qui (Il Paradosso del Deadlock)

```
int contatore = 0;
pthread_mutex_t lock;

void incrementa() {
    pthread_mutex_lock(&lock);   // Prende il lock
    contatore++;
    pthread_mutex_unlock(&lock);
}
```

Questa funzione è **MT-Safe** ma **NON** Async-Signal-Safe . Scenario:

1. Il `main` chiama `incrementa()` prende il **Lock**
2. Sta per fare `contatore++` ... ma **ARRIVA UN SEGNALE!**
3. Il Kernel congela l'esecuzione, salta nell'Handler
4. L'Handler chiama `incrementa()` di nuovo
5. Tenta `pthread_mutex_lock` il Mutex è **già chiuso** (da sé stesso!)
6. L'Handler si **blocca in attesa**... ma l'unico che può sbloccare è il `main` ... che è congelato
7. **DEADLOCK — il programma è bloccato per sempre**

> *[!CAUTION]*
> *`printf` e `malloc` **NON sono Async-Signal-Safe!** Non chiamarle **MAI** dentro un Signal Handler. Solo pochissime funzioni lo sono (es. `write` , `read` , `_exit`).*

Tema 49: Funzioni Rientranti (Reentrant / `_r`) e Regola d'Oro dei Segnali

Reentrancy — Il trucco definitivo

Definizione: Una funzione è **rientrante** se non usa **nessuna** variabile condivisa (né globale, né statica). Tutto è in variabili **locali** (Stack) o passato come **parametro**.

Proprietà	**Conseguenza**
Reentrant automaticamente	**MT-Safe** + **Async-Signal-Safe**
MT-Safe (con mutex)	**NON** garantisce Async-Signal-Safe

L'esempio `strtok` vs `strtok_r`

```
// strtok: variabile static interna    NÉ MT-Safe NÉ Async-Safe
char *strtok(char *str, const char *delim);

// strtok_r: lo stato è passato dal chiamante    Reentrant!
char *strtok_r(char *str, const char *delim, char **saveptr);
```

`strtok_r` chiede al programmatore di passare `saveptr` (puntatore locale). **Niente variabili statiche, niente conflitti.**

`qsort` vs `qsort_r`

Se la funzione di confronto ha bisogno di dati esterni, con `qsort` saresti costretto a usare una variabile globale. `qsort_r` aggiunge un **5° parametro** per passare un argomento extra direttamente alla funzione di confronto, senza globali.

La Regola d'Oro dei Signal Handler (Risposta da 30 e Lode)

> *[!IMPORTANT]*
> *"Come si scrivono i Signal Handler in C?"*

1. Devono essere **cortissimi**
2. **Mai** chiamare `printf`, `malloc` o funzioni complesse
3. L'unica azione sicura: modificare una variabile `volatile sig_atomic_t`

```
volatile sig_atomic_t flag_uscita = 0;  // Intero atomico garantito

void handler(int signum) {
    flag_uscita = 1;  // Setta il flag e basta!
}

// Nel main:
while (!flag_uscita) {
    // ... lavoro normale ...
}
// Qui fai la pulizia (fclose, free, ecc.)
```

> *[!TIP]*
> *`sig_atomic_t` è un tipo intero speciale **garantito atomico** per le operazioni di lettura/scrittura, anche in contesto di segnali. Il `main` lo legge in sicurezza.*

Tema 50: Il Thread Gestore dei Segnali (Best Practice Assoluta)

> *[!IMPORTANT]*
> ***Domanda d'esame classica:*** *"Come si gestiscono i segnali in modo pulito in un programma multi-thread?"*

I 4 passi fondamentali

Passo 1: Nascondi tutto nel `main`

```
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT);
sigaddset(&mask, SIGTERM);
pthread_sigmask(SIG_BLOCK, &mask, NULL);  // Blocca i segnali
```

Passo 2: Crea i Thread Lavoratori

I thread ereditano la Signal Mask nascono con i segnali **già bloccati**. Possono usare `printf` e Mutex in totale sicurezza.

Passo 3: Crea il Thread Gestore

```
pthread_create(&tid_gestore, NULL, thread_gestore, &mask);
```

Passo 4: Usa `sigwait` nel Gestore

```
void *thread_gestore(void *arg) {
    sigset_t *mask = (sigset_t *)arg;
    int sig;

    while (1) {
        sigwait(mask, &sig);  // Si BLOCCA finché non arriva un segnale

        switch (sig) {
            case SIGINT:
                printf("Ricevuto SIGINT, chiudo...\n");  // printf OK qui!
                // Pulizia: shm_unlink, fclose, free...
```

```
            exit(EXIT_SUCCESS);
        case SIGTERM:
            printf("Ricevuto SIGTERM\n");
            exit(EXIT_SUCCESS);
      }
    }
}
```

> *[!TIP]*
> ***La magia di `sigwait` :*** *Non lancia handler asincroni! "Ruba" il segnale dallo stato Pending e restituisce il suo numero in una variabile. La gestione diventa* ***sincrona e sicura*** *— puoi usare `printf` , `malloc` , Mutex, tutto!*

Tema 51: `pthread_sigmask` e le Macro per le Maschere

> *[!IMPORTANT]*
> ***Pagina man richiesta dal prof:*** *Nei multi-thread si* ***DEVE*** *usare `pthread_sigmask` (non `sigprocmask`), perché la maschera è* ***privata per ogni thread****.*

```
#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
```

Il parametro `how`

Valore	Operazione
`SIG_BLOCK`	**Unione (OR):** Aggiungi questi segnali a quelli già bloccati.
`SIG_UNBLOCK`	**Sottrazione:** Rimuovi questi segnali da quelli bloccati.
`SIG_SETMASK`	**Sostituzione:** Rimpiazza la vecchia maschera con questa nuova.

Come creare la maschera `sigset_t`

```
sigset_t set;
sigemptyset(&set);          // 1. Svuota tutto a zero
sigaddset(&set, SIGINT);    // 2. Alza il bit di SIGINT
sigaddset(&set, SIGTERM);   // 3. Alza il bit di SIGTERM
```

> *[!WARNING]*
> ***Non** usare operatori bit-a-bit del C! Usa **solo** le funzioni* `sigemptyset` / `sigaddset` / `sigfillset` .

Tema 52: Dettagli di Architettura dei Segnali

A. Perché `sig_atomic_t` e non un normale `int` ?

Se un'architettura ha interi a 64 bit ma registri a 32 bit, scrivere un intero richiede **due operazioni CPU**. Se il segnale arriva **in mezzo** alle due l'intero è **corrotto**.

`volatile sig_atomic_t` garantisce che la scrittura avvenga in un **singolo ciclo di clock** (atomica). Impossibile interromperla a metà.

B. Ereditarietà: `fork` vs `exec` vs `pthread_create`

Operazione	Signal Handler Array	Signal Mask
`fork()`	**Eredita** (copia)	**Eredita** (copia)
`pthread_create()`	**Condivide** (stesso array)	**Eredita** dal thread creatore
`exec()`	**Reset a `SIG_DFL`**	**Mantiene** la maschera

> *[!CAUTION]*
> ***`exec()` distrugge gli handler!** Sostituendo il programma, i codici delle funzioni handler vengono eliminati il Kernel resetta tutto a `SIG_DFL` . **Eccezione:** i segnali su `SIG_IGN` restano ignorati.*

C. L'anomalia di `SIGCHLD`

`SIGCHLD` è il segnale inviato automaticamente al Padre quando un Figlio termina.

> *[!IMPORTANT]*
> *Se 5 figli terminano quasi contemporaneamente, il padre deve saperlo per fare 5* `wait` *(evitando zombie). La best practice è usare* `waitpid` *in un ciclo nell'handler:*

```
void handler_sigchld(int sig) {
   // Mieti TUTTI i figli morti in un colpo solo
   while (waitpid(-1, NULL, WNOHANG) > 0)
```

```
    ; // Continua finché ci sono zombie da raccogliere
}
```

> *[!NOTE]*
> `WNOHANG` *fa sì che* `waitpid` ***non si blocchi****: se non ci sono più figli morti, ritorna subito* `0` *o* `-1` *.*

Tema 53: Design Patterns per Segnali Specifici

`SIGINT` / `SIGTERM` — Pulizia prima dell'uscita

Catturali **solo** per fare cleanup: `shm_unlink` , `fclose` , chiudere socket, svuotare buffer. Poi `exit()` .

`SIGSEGV` — Segmentation Fault

Puoi catturarlo per un log formattato, ma:

> *[!CAUTION]*
> *L'ultima istruzione dell'handler* ***DEVE*** *essere* `exit(EXIT_FAILURE)` *.* ***Mai*** *ritornare al* `main` *dopo un segfault: la memoria è corrotta e crasherà di nuovo all'infinito.*

`SIGPIPE` — Il killer silenzioso nell'IPC

Se scrivi su una pipe/socket il cui lettore è morto il processo riceve `SIGPIPE` **l'azione default lo uccide!**

```
// All'inizio del main di OGNI programma client-server:
signal(SIGPIPE, SIG_IGN);  // Ignora SIGPIPE
```

> *[!TIP]*
> *Se ignori* `SIGPIPE` *, la* `write()` *semplicemente fallisce con* `-1` *e* `errno = EPIPE` *. Il server* ***continua a funzionare*** *e gestisce l'errore pacificamente.*

Tema 54: Il Paradigma Definitivo — `sigaction` vs `sigwait`

> *[!IMPORTANT]*
> *Riassunto da esporre se il prof chiede: "Come si gestiscono i segnali in multi-threading?"*

Approccio	Meccanismo	Rischi
Vecchio (`sigaction`)	Handler asincrono, interrompe il thread	Async-Signal-Unsafe, variabili globali
Nuovo (Thread + `sigwait`)	Gestione sincrona, thread dedicato	**Nessuno** — `printf` / `malloc` OK

`sigwait` — Prototipo

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
```

Parametro	Tipo	Descrizione
`set`	`const sigset_t *`	Maschera dei segnali da aspettare (**devono essere bloccati!**).
`sig`	`int *`	Variabile dove viene scritto il numero del segnale arrivato.

> *[!TIP]*
> *Il thread gestore **dorme** su `sigwait` . Quando arriva un segnale, si sveglia e lo gestisce **riga per riga**, in modo sincrono. Niente handler asincroni, niente variabili globali, niente interruzioni.*

Tema 55: Segnali Standard vs Segnali Real-Time (RT)

In Linux ci sono **64 segnali**.

Range	Tipo	Accodamento	Payload
1-31	**Standard**	Fusi in 1 (Pending Bitmap)	No
32-64	**Real-Time**	**Accodati** (coda vera)	Intero o puntatore

Ordine di Precedenza (Domanda fine!)

Se ci sono segnali pendenti sia standard che RT:

1. **Prima gli Standard** (1-31): vengono sempre consegnati prima
2. **Poi gli RT per priorità**: numero più basso = priorità più alta (`SIGRTMIN` prima di `SIGRTMAX`)

Tema 56: Inviare e Ricevere Payload — `sigqueue` e `sigwaitinfo`

Inviare con `sigqueue`

```
#include <signal.h>
int sigqueue(pid_t pid, int sig, const union sigval value);
int pthread_sigqueue(pthread_t thread, int sig, const union sigval value);
```

Il Mistero della `union sigval`

```
union sigval {
    int   sival_int;    // 4 byte
    void *sival_ptr;    // 8 byte
};  // Dimensione totale: 8 byte (il campo maggiore)
```

Differenza `struct` vs `union`

Tipo	Campi in memoria	Dimensione
`struct`	Uno **di fianco** all'altro (4 + 8 = 12 byte)	Somma dei campi
`union`	**Stessa posizione** in memoria	Il campo maggiore

> *[!CAUTION]*
> ***"Mettersi d'accordo":*** *Se il Processo A scrive `value.sival_int = 42` e il Processo B legge `value.sival_ptr` , interpreterà quei bit come indirizzo di memoria* ***Segmentation Fault!*** *I programmatori devono concordare: "Usiamo solo `sival_int` ". (Un puntatore tra processi diversi è inutile: hanno spazi di memoria virtuale separati).*

Ricevere con `sigwaitinfo` (La pagina MAN dell'esame!)

> *[!IMPORTANT]*
> *Il prof ti chiede di commentare questa pagina man!*

```
#include <signal.h>
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
```

Parametro	Tipo	Descrizione
Ritorno	`int`	Il numero del segnale ricevuto.
`set`	`const sigset_t *`	Maschera dei segnali da aspettare.
`info`	`siginfo_t *`	Struct dove il Kernel scrive **tutti i dettagli** del segnale.

Campi utili di `siginfo_t`

Campo	Tipo	Contenuto
`info->si_signo`	`int`	Numero del segnale arrivato.
`info->si_pid`	`pid_t`	**PID di chi ha mandato il segnale!**
`info->si_value.sival_int`	`int`	Il **Payload** inviato con `sigqueue`.

> *[!NOTE]*
> `sigwait` *riceve solo il numero del segnale.* `sigwaitinfo` *riceve* ***anche*** *il payload e il PID del mittente. Per i segnali RT con dati, usa sempre* `sigwaitinfo`.

Tema 57: `raise` e `volatile` (Dettagli Avanzati)

`raise` — Inviare un segnale a sé stessi

```
#include <signal.h>
int raise(int sig);
// Equivale a: pthread_kill(pthread_self(), sig);
```

`volatile` — Perché è obbligatorio (Spiegazione completa)

Il compilatore GCC con `-O2` / `-O3` ottimizza aggressivamente:

```
int flag = 0;
while (flag == 0) { /* aspetta */ }
```

GCC pensa: *"Il ciclo non modifica `flag`. Lo carico in un registro CPU velocissimo e leggo sempre quello."*

Ma se un Signal Handler fa `flag = 1` scrive nella **RAM**, mentre il `while` legge dal **registro** (che è ancora `0`). **Il programma non si ferma mai!**

```
volatile sig_atomic_t flag = 0;  // SOLUZIONE
```

> *[!CAUTION]*
> *`volatile` è un ordine tassativo al compilatore: "Non ottimizzare questa variabile. Qualcuno potrebbe modificarla di nascosto. Vai a leggerla dalla RAM ad ogni singolo giro del ciclo."*

Senza `volatile`	Con `volatile`
GCC legge dal **registro** (cache)	GCC legge dalla **RAM** (sempre aggiornata)
Handler scrive nella RAM ignorato	Handler scrive nella RAM il `while` lo vede
Bug silenzioso — programma bloccato	Funziona correttamente

Tema 58: Codici Completi — Graceful Shutdown con Segnali

> *[!IMPORTANT]*
> *Due esempi pronti per l'esame e il progetto, commentati riga per riga.*

Approccio 1: Asincrono con `sigaction` (Singolo Thread)

Regole dimostrate: `volatile sig_atomic_t`, nessuna `printf` nell'handler, `sa_mask`.

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

```c
#include <signal.h>
#include <unistd.h>

// 1. Variabile globale atomica per comunicare handler    main
volatile sig_atomic_t flag_uscita = 0;

// 2. Handler CORTISSIMO: alza la bandierina e basta
void gestore_sigint(int signum) {
    // NESSUNA printf QUI! Non è Async-Signal-Safe.
    flag_uscita = 1;
}

int main() {
    struct sigaction sa;
    sa.sa_handler = gestore_sigint;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGTERM);  // Blocca SIGTERM durante l'handler
    sa.sa_flags = 0;

    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("Errore sigaction");
        exit(EXIT_FAILURE);
    }

    printf("Premi Ctrl+C per uscire in modo pulito...\n");

    // 3. Main Loop: controlla la bandierina
    while (flag_uscita == 0) {
        printf("Lavoro in corso...\n");
        sleep(2);
    }

    // Ctrl+C premuto    pulizia
    printf("\n[MAIN] Bandierina vista! Pulizia ed uscita.\n");
    // ... fclose, free, shm_unlink ...
    return EXIT_SUCCESS;
}
```

Approccio 2: Sincrono con Thread Dedicato e `sigwait` (Multi-Thread)

Regole dimostrate: Maschera **prima** dei thread, ereditarietà, `printf` sicura nel gestore.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <pthread.h>
#include <unistd.h>

// Il Thread Gestore dei Segnali
void* thread_gestore_segnali(void* arg) {
    sigset_t *maschera = (sigset_t*) arg;
    int segnale_ricevuto;

    printf("[THREAD GESTORE] In attesa di segnali...\n");

    while (1) {
        // Si addormenta qui. Consuma 0 CPU.
        int err = sigwait(maschera, &segnale_ricevuto);
        if (err != 0) { perror("Errore sigwait"); continue; }

        // Contesto SINCRONO    printf, mutex, malloc OK!
        if (segnale_ricevuto == SIGINT) {
            printf("\n[GESTORE] SIGINT ricevuto! Spegnimento pulito...\n");
            // Pulizia: shm_unlink, fclose, free...
            exit(EXIT_SUCCESS);
        }
    }
    return NULL;
}

int main() {
    sigset_t maschera_segnali;
    pthread_t thread_id;

    // 1. Preparo la maschera
    sigemptyset(&maschera_segnali);
    sigaddset(&maschera_segnali, SIGINT);
```

```
    sigaddset(&maschera_segnali, SIGTERM);

    // 2. BLOCCO i segnali nel main (PRIMA di creare qualsiasi thread!)
    if (pthread_sigmask(SIG_BLOCK, &maschera_segnali, NULL) != 0) {
        perror("Errore pthread_sigmask");
        exit(EXIT_FAILURE);
    }

    // 3. Creo il thread gestore (eredita la maschera    segnali bloccati)
    if (pthread_create(&thread_id, NULL, thread_gestore_segnali,
                       (void*)&maschera_segnali) != 0) {
        perror("Errore pthread_create");
        exit(EXIT_FAILURE);
    }

    // 4. Da qui: crea thread worker o lavora
    printf("[MAIN] Al sicuro dai segnali. Lavoro...\n");
    while (1) {
        sleep(3);
        printf("[MAIN] Sto lavorando tranquillo...\n");
    }

    return 0;  // Terminerà via exit() nel thread gestore
}
```

Quale scegliere per il progetto?

> *[!IMPORTANT]*
> ***Senza dubbio l'Approccio 2.*** *Se porti un server multi-thread con* `sigaction` *e variabili globali atomiche, il prof farà mille domande su interleaving e Async-Signal-Safety. Con il Thread dedicato +* `sigwait` *, dimostri di aver capito l'architettura avanzata di Linux.*

Criterio	**Approccio 1 (`sigaction`)**	**Approccio 2 (`sigwait`)**
Complessità	Semplice	Leggermente più articolato
Sicurezza Multi-Thread	Rischiosa	Totale
`printf` nel gestore?	Vietata	Permessa

`malloc` nel gestore?	Vietata	Permessa
Variabili globali?	Obbligatorie (`volatile`)	Non necessarie
Voto atteso al progetto	Sufficienza	**30 e Lode**

Schema di Sintesi per l'Esame

Se il professore ti chiede di confrontare, ricorda questo schema mentale:

Meccanismo	**Solo tra imparentati**	**Per tutti i processi**
Comunicazione	`pipe` (senza nome)	`mkfifo` (pipe con nome / FIFO)
Sincronizzazione	Semafori senza nome / Mutex	**Semafori con nome** (`sem_open`)
Dati condivisi	`pipe` (copia via kernel)	**Shared Memory** (`shm_open` + `mmap`)

Parte III — Python

Tema 59: Dizionari, `__eq__` e `__hash__`

> *[!IMPORTANT]*
> ***Domanda d'esame:*** *"Metodo eq e hash in Python"*

I dizionari (`dict`) e i `set` funzionano con **Tabelle di Hash**. Per usare un oggetto come chiave, deve essere **immutabile**.

Tipo	**Chiave di dizionario?**	**Perché?**
`list`	No	Mutabile (`append` la cambia)
`tuple`	Sì	Immutabile
Classe	Se definisci `__eq__` e `__hash__`	Devi dire a Python come fare

Override dei metodi magici

```
class Studente:
    def __init__(self, matricola, nome):
        self.matricola = matricola
        self.nome = nome

    def __repr__(self):
        return f"{self.nome} ({self.matricola})"

    def __eq__(self, other):
        if not isinstance(other, Studente):
            return False
        return self.matricola == other.matricola

    def __hash__(self):
        return hash(self.matricola)

# Ora posso usarlo come chiave!
dizionario_voti = {Studente("12345", "Mario"): 30}
```

> *[!CAUTION]*
> ***Regola d'oro:*** *Se due oggetti sono uguali secondo* `__eq__` *, il loro* `__hash__` ***DEVE*** *restituire lo stesso numero. Altrimenti il dizionario si comporta in modo imprevedibile.*

Tema 60: Lettura File di Testo e Parsing

> *[!IMPORTANT]*
> ***Domanda d'esame:*** *"Scrivere una classe per salvare i dati di un file di testo"*

Perché `with open()` e non `f = open()`?

Il costrutto `with` (Context Manager) **chiude automaticamente** il file alla fine del blocco, anche in caso di eccezione.

```
studenti = []

try:
```

```
    with open("dati.txt", "r") as f:
        for linea in f:
            # .split() divide per spazi e rimuove \n automaticamente
            dati = linea.split()

            if len(dati) == 3:
                matricola, nome, cognome = dati[0], dati[1], dati[2]
                studenti.append(Studente(matricola, nome))
except FileNotFoundError:
    print("Il file non esiste!")
```

> *[!TIP]*
> `.split()` *senza argomenti divide usando* ***qualsiasi spazio bianco*** *(spazi, tab, newline) e rimuove gli spazi vuoti. È più robusto di* `.split(" ")` *.*

Tema 61: Ordinamento — `sort()` vs `sorted()`

Aspetto	**lista.sort()**	**sorted(lista)**
Modifica	**In-place** (cambia la lista)	Crea una **copia** ordinata
Ritorno	`None`	Nuova lista
Funziona su	Solo `list`	Qualsiasi iterabile (tuple, dict...)

Ordinamento naturale delle tuple

```
lista = [(3, "Zebra"), (1, "Alpaca"), (2, "Gatto")]
sorted(lista)  #    [(1, 'Alpaca'), (2, 'Gatto'), (3, 'Zebra')]
```

Confronta prima il 1° elemento. In caso di pareggio, il 2°.

Ordinamento custom con `key` (Keyword Argument)

```
# Ordina per nome dell'animale (indice 1)
sorted(lista, key=lambda x: x[1])
```

```
# Ordina oggetti Studente per nome
studenti.sort(key=lambda s: s.nome)
```

Tema 62: Lo Shebang e `sys.argv`

Rendere uno script eseguibile

1. Prima riga del file (**Shebang**): `#!/usr/bin/env python3`
2. Dare i permessi: `chmod +x script.py`
3. Lanciare: `./script.py`

> *[!NOTE]*
> *Perché `env python3` e non `/usr/bin/python3`? Perché Python potrebbe essere installato altrove. `env` cerca nell' `$PATH` e trova l'eseguibile giusto.*

`sys.argv` — Argomenti da riga di comando

```
import sys

# Se chiamo: ./script.py file.txt 42
# sys.argv[0]    "./script.py"
# sys.argv[1]    "file.txt"
# sys.argv[2]    "42"  (STRINGA! Devi fare int() se ti serve un numero)

argc = len(sys.argv)
if argc != 3:
    print(f"Uso: {sys.argv[0]} <file> <numero>")
    sys.exit(1)
```

Tema 63: I Moduli `os` e `os.path`

`os.listdir` — Elencare il contenuto di una cartella

```
import os

cartella = "/tmp"
```

```
if os.path.isdir(cartella):
    for nome in os.listdir(cartella):
        # IMPORTANTE: os.listdir restituisce SOLO i nomi, senza percorso!
        percorso_completo = os.path.join(cartella, nome)
        print(percorso_completo)
```

`os.path.join` — Mai concatenare stringhe per i percorsi!

> *[!WARNING]*
> ***Non scrivere mai*** *`cartella + "/" + file`. Su Windows i percorsi usano `\`, su Linux `/`.* *`os.path.join`* *mette lo slash corretto automaticamente.*

```
#   SBAGLIATO:
percorso = cartella + "/" + file

#   CORRETTO:
percorso = os.path.join(cartella, file)
```

Tema 64: Permessi sulle Directory — La Trappola R vs X

> *[!IMPORTANT]*
> *Si collega alla domanda d'esame su* ***UMASK*** *e permessi.*

Permesso	Su un File	Su una Directory
R	Leggere il contenuto	Elencare i nomi (`ls`) ma **NON** accedere ai file
X	Eseguire il programma	**Attraversare** la directory (`cd` , usarla in un path)

> *[!CAUTION]*
> *Avere* ***R senza X*** *su una cartella è quasi inutile: puoi vedere i nomi dei file ma non puoi aprirne nessuno!*

Verifica con `os.access`

```
import os
```

```
percorso = "/cartella_segreta"
# Verifico R (ls) e X (cd) con OR bit-a-bit
if os.access(percorso, os.R_OK | os.X_OK):
    print("Posso entrare ed elencare i file!")
else:
    print("Permesso negato.")
```

Tema 65: Link Simbolici (Soft Links)

Un link simbolico (`ln -s destinazione nome_link`) è un file speciale che contiene una **stringa testuale** puntante a un altro file. Equivalente del "Collegamento" Windows.

Funzione	Cosa fa
`os.path.islink(path)`	`True` se il file è un link simbolico
`os.path.realpath(path)`	Risolve il link (anche concatenati) percorso **assoluto reale**

```
import os

percorso = "/tmp/mio_link"
if os.path.islink(percorso):
    reale = os.path.realpath(percorso)
    print(f"Il link punta a: {reale}")
```

> *[!TIP]*
> `realpath` *è utilissimo per* ***non processare lo stesso file due volte*** *quando si scansiona una directory che contiene link.*

Tema 66: `subprocess` — Eseguire Comandi di Sistema da Python

Il comando `cmp` (Compare)

Confronta due file byte per byte. Exit status: `0` = identici, `1` = diversi.

`subprocess.run` — Uso corretto

```
import subprocess

# LISTA di stringhe, NON una stringa unica (evita Shell Injection!)
comando = ["cmp", "file1.txt", "file2.txt"]

risultato = subprocess.run(comando, capture_output=True, text=True)

if risultato.returncode == 0:
    print("I file sono identici.")
else:
    print("I file sono diversi.")
    print("Errore:", risultato.stderr)
```

Parametro	Cosa fa
`capture_output=True`	Non stampa a schermo, salva in `risultato.stdout` / `.stderr`
`text=True`	Converte i byte binari del SO in **stringhe** leggibili
`risultato.returncode`	Exit status del comando (0 = successo)

> *[!WARNING]*
> ***Non passare una stringa!*** *`subprocess.run("cmp file1 file2")` è sbagliato. Passa sempre una* ***lista****:* *`["cmp", "file1", "file2"]`.*

Tema 67: Le Classi in Python — `self`, `__init__`, `__str__`

`self` — L'equivalente di `this` (Java/C++)

In Python non si dichiarano le variabili all'inizio della classe. `self` è l'unico modo per dire *"Sto modificando l'attributo di QUESTO specifico oggetto"*. Deve essere il **primo parametro** di tutti i metodi.

I metodi magici

```
class Giocatore:
    def __init__(self, nome, squadra):   # Costruttore (chiamato automaticamente)
        self.nome = nome
        self.squadra = squadra
```

```
    def __str__(self):                  # Invocato da print(oggetto)
        return f"{self.nome} (Gioca in: {self.squadra})"
```

Metodo	Quando viene chiamato	Se non lo definisci...
`__init__`	`p = Giocatore("Totti", "Roma")`	Oggetto creato senza attributi
`__str__`	`print(p)` o `str(p)`	Stampa `<__main__.Giocatore object at 0x7f8...>`
`__repr__`	`repr(p)` o nel REPL interattivo	Come `__str__` ma per debugging

Tema 68: La Teoria Completa di `__eq__` e `__hash__` (Collisioni)

> *[!IMPORTANT]*
> ***Domanda d'esame definitiva:*** *"Come interagiscono eq e hash?"*

Il problema dell'uguaglianza

Senza override di `__eq__`, due oggetti sono uguali **solo se sono lo stesso oggetto in memoria** (stesso `id()`):

```
g1 = Giocatore("Totti", "Roma")
g2 = Giocatore("Totti", "Roma")
g1 == g2  #    False (senza __eq__)!  Sono due oggetti fisici diversi
```

Quando viene invocato `__eq__` (invisibilmente)

1. `if g1 == g2:` — confronto diretto
2. `if g1 in lista:` — Python scorre la lista e chiama `__eq__` per ogni elemento

L'hash con tuple (trucco per oggetti composti)

```
class Giocatore:
    def __eq__(self, other):
        if not isinstance(other, Giocatore):
            return False
        return self.nome == other.nome and self.squadra == other.squadra

    def __hash__(self):
        return hash((self.nome, self.squadra))  # Tupla    hash unico!
```

La Regola d'Oro e le Collisioni (Risposta da 30!)

> *[!IMPORTANT]*
> ***Il Contratto Indissolubile:***
> *Se* `a == b` *è* `True` *(secondo* `__eq__` *), allora* `hash(a) == hash(b)` ***DEVE*** *essere* `True` *.*

> *[!NOTE]*
> ***Il viceversa NON vale!*** *Se* `hash(a) == hash(b)` *,* ***non è detto*** *che* `a == b` *. Questo si chiama* ***Collisione****.*

Come si comporta l'HashSet in caso di collisione

```
Inserimento di un oggetto in un Set:

1. Python calcola hash(oggetto)
2. Guarda nel Bucket corrispondente
        Bucket VUOTO    inserisce l'oggetto
        Bucket OCCUPATO (stesso hash)    NON si fida!
           Chiama __eq__ tra i due oggetti
              __eq__    True     DUPLICATO    lo scarta
              __eq__    False    COLLISIONE    li salva entrambi
           (internamente usa una lista concatenata nel bucket)
```

> *[!CAUTION]*
> *Se usi* `nome` *e* `squadra` *in* `__eq__` *,* ***devi*** *usare* `nome` *e* `squadra` *in* `__hash__` *.*
> *Se non rispetti questa regola, le tabelle di hash si rompono e* ***perdi dati*** *nei dizionari.*

Tema 69: OOP Avanzata — `__repr__`, Confronti e `__add__`

`__str__` vs `__repr__`

Metodo	Per chi?	Quando?	Cosa deve restituire?
`__str__`	Per l'**utente**	`print(obj)`, `str(obj)`	Descrizione informale e leggibile
`__repr__`	Per il **programmatore**	`repr(obj)`, console interattiva, liste	Stringa che **ricrea l'oggetto** se copiata

```
def __repr__(self):
    return f"Punto({self.x}, {self.y})"  # Copiabile nel codice!
```

Rich Comparison Operators (Ordinamento)

Metodo	Operatore	Significato
`__lt__`	`<`	Less Than
`__le__`	`<=`	Less or Equal
`__gt__`	`>`	Greater Than
`__ge__`	`>=`	Greater or Equal

> *[!TIP]*
> *Per usare `lista.sort()`, basta definire `__lt__` e `__eq__`. Python **deduce tutti gli altri** operatori da questi due!*

Overloading di `+` con `__add__`

```
class Prodotto:
    def __init__(self, nome, prezzo):
        self.nome = nome
        self.prezzo = prezzo

    def __lt__(self, altro):
```

```
        return self.prezzo < altro.prezzo

    def __add__(self, altro):
        return self.prezzo + altro.prezzo

    def __repr__(self):
        return f"Prodotto('{self.nome}', {self.prezzo})"

p1 = Prodotto("Mela", 2)
p2 = Prodotto("Pane", 3)
print(p1 + p2)  #    5 (chiama __add__ automaticamente)
```

Tema 70: Il GIL e il Garbage Collector (Domanda da 30 e Lode!)

> *[!IMPORTANT]*
> ***Domanda d'esame:*** *"Parlami dei Thread in Python" — Il prof vuole sentire questa storia.*

1. Il Garbage Collector e il Reference Counting

In Python **non serve la** `free()` . Ogni oggetto ha un **contatore** invisibile:

```
a = Oggetto()  # contatore = 1
b = a          # contatore = 2
del a          # contatore = 1
del b          # contatore = 0    DISTRUTTO, RAM liberata
```

2. Il problema del Multi-Threading

Se due thread modificano lo stesso contatore **Race Condition** contatore sballa **Memory Leak** o **Segfault**.

3. Le due soluzioni possibili

Soluzione	Come funziona	Problema

A. Granular Locking	Un Mutex per ogni oggetto	Milioni di lock/unlock **lentissimo** + Deadlock
B. GIL (scelta di Python)	Un **singolo** Mutex per tutto l'interprete	Solo 1 thread esegue alla volta

4. Cos'è il GIL?

Global Interpreter Lock — Un singolo, gigantesco Mutex.

> *"Solo un thread alla volta può eseguire Bytecode Python."*

100 thread + 8 core **solo 1 core lavora**, gli altri 99 aspettano il turno.

5. Allora i thread sono inutili?

Tipo di lavoro	Thread utili?	Perché?
CPU-Bound (calcoli)	Inutili	Peggiorano le prestazioni (tempo perso a passarsi il GIL)
I/O-Bound (rete, disco)	**Utilissimi!**	Python **rilascia il GIL** durante l'attesa I/O altri thread lavorano

---

Tema 71: I Thread in Python — Modulo `threading`

L'approccio classico ricalca `pthread_create` / `pthread_join` del C.

```
import threading
import time

def lavoratore(nome):
    print(f"Thread {nome} partito.")
    time.sleep(2)  # Simula lavoro I/O bound
    print(f"Thread {nome} finito.")

# 1. Creazione (≈ preparare pthread_create)
t1 = threading.Thread(target=lavoratore, args=("Uno",))
```

```
# 2. Avvio (il thread parte ORA)
t1.start()

# 3. Attesa (≈ pthread_join — il main aspetta qui)
t1.join()
```

Passo	Python	C (POSIX)
Creazione	`threading.Thread(target=...)`	`pthread_create(&t,...)`
Avvio	`t.start()`	(incluso in create)
Attesa	`t.join()`	`pthread_join(t, NULL)`

Tema 72: `ThreadPoolExecutor` — L'Approccio Moderno

Creare/distruggere thread consuma risorse. Il **Pool** crea un gruppo di thread operai riutilizzabili.

`with` applicato al Pool

Come `with open()` chiude il file, `with ThreadPoolExecutor` fa la **`join` automatica** di tutti i thread all'uscita!

`map` vs `submit`

Metodo	Uso	Quando?
`map`	Assegna **automaticamente** ogni elemento a un thread	Stessa funzione su molti dati
`submit`	Lancia **un singolo** task (serve un `for` per N)	Task diversi o controllo fine

```
import concurrent.futures

def quadrato(n):
    return n * n

numeri = [1, 2, 3, 4, 5]
```

```
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    risultati = executor.map(quadrato, numeri)

# Usciti dal 'with'    JOIN automatica di tutti i thread!
print(list(risultati))  #    [1, 4, 9, 16, 25]
```

Tema 73: Il Modulo `logging` (Thread-Safe)

> *[!WARNING]*
> ***Mai usare `print` in programmi multi-thread!*** *Se due thread stampano contemporaneamente, i testi si mescolano.*

`logging` è **Thread-Safe** (ha mutex interni) e introduce **livelli di severità**:

Livello	Numero	Quando usarlo
`DEBUG`	10	Info per il programmatore
`INFO`	20	Il programma procede regolarmente
`WARNING`	30	Qualcosa è strano, ma funziona
`ERROR`	40	Una funzione ha fallito
`CRITICAL`	50	Il programma sta per esplodere

```
import logging

# Configuro: livello minimo INFO    ignora DEBUG
logging.basicConfig(level=logging.INFO,
                    format='%(threadName)s - %(message)s')

logging.debug("Non verrà stampato (troppo basso).")
logging.info("Server avviato sulla porta 8080.")
logging.warning("Attenzione, memoria quasi piena.")
```

> *[!TIP]*
> *Impostando `level=logging.INFO`, vengono stampati **solo** i messaggi da `INFO` in su.*
> *I `DEBUG` vengono ignorati. In produzione si alza il livello a `WARNING`.*