0 ALLOCAZIONE DELLA MEMORIA 4 marzo

Un *programma eseguibile* è costituito da un insieme di:

- **istruzioni macchina:** *il cui significato è cablato nel processore specifico per cui il programma è stato creato*
- **dati**
- valori di **configurazione** e **controllo** rappresentati come sequenze di numeri binari.

NOTA: Ad ogni byte corrisponde una ed una sola istruzione macchina.
NOTA: La presentazione di dati e informazioni di controllo è ricavata dalle istruzioni macchina e dal sistema operativo.

Il **programma** vive in un file sul disco ma per poter eseguire il programma, il file deve essere trasferito **all'interno della RAM** dove poi il processore riesce a prendere le istruzioni. *(exe rappresentano i binari eseguibili)*

- *Nei sistemi piccoli il passaggio avviene grazie ad un HW specifico che memorizza il necessario in flash in maniera semipermanente*
- Nei sistemi grandi, il loader trasferisce il contenuto del file eseguibile dal disco alla RAM, per permettere l'esecuzione.

*NOTA: il **loader** è un modulo del sistema operativo*

Quando il programma è in memoria, può essere eseguito.

ESECUZIONE:

processore preleva un'istruzione da una certa cella di memoria, capisce il dato e fa quel che c'è scritto → CICLO:

- **Fetch**: istruzione da memoria a processore
- **Decode**: istruzione prelevata viene trasformata in comandi da eseguire
- **Execute**: comandi vengono eseguiti

Processore fa riferimento ad una specifica cella in cui sta l'istruzione, indicando il registro:

- **IP**: registro presente nella CPU che ricorda l'indirizzo della prossima istruzione da eseguire
- Altri registri vengono usati per ricordare indirizzi relativi ai dati o alle strutture di controllo che il programma utilizza (**SP**: Stack pointer ricorda dove certi dati devono essere messi e dove altri vanno prelevati)

Esecuzione del programma avviene nello spazio di indirizzamento *(tutto ciò che abbiamo nel file exe finisce in questa piccola porzione della RAM).*

SPAZIO DI INDIRIZZAMENTO:

Sottoinsieme di celle indirizzabili gestite dal sistema operativo
Possiamo vedere lo spazio di indirizzamento come un Array di byte consecutivi.

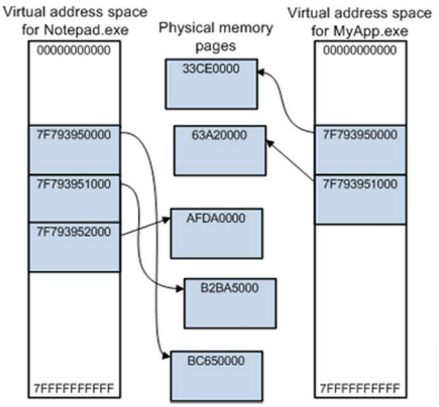
- Può essere letto 1-2-4-8 byte alla volta *(in base al parallelismo del processore)*
- Spazi di indirizzamento **non sono completi e non sono contigui**, il programma ha **l'illusione** di averli tutti ma in realtà ne ha solo alcuni
- se si legge fuori dalla porzione a disposizione → segmentation fault.

NOTA: nel x64 lo spazio di indirizzamento è limitato a 48 bit → 2^{48} celle ma in realtà solo una piccola parte di indirizzi è effettivamente presente.

ILLUSIONE E MMU

Indirizzi che il programma maneggia, non corrispondono al posto in cui il processore legge/scrive in memoria:

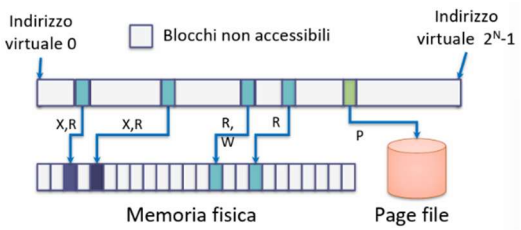
- *Io vedo indirizzi che vanno tra 0 e N-1. In realtà a livello HW si va in indirizzi differenti.*



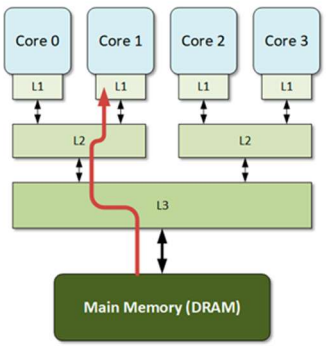
MMU traduce gli indirizzi virtuali in indirizzi fisici. Questo fa in modo che i programmi possano lavorare contemporaneamente senza darsi fastidio. Dunque, **il sistema operativo mantiene per ogni processo in esecuzione, la tabella di corrispondenza degli indirizzi**.

*Il mapping virtuale-fisico è soggetto ad alcune annotazione, il sistema operativo mi segna anche a quali **vincoli** quel mapping è soggetto (es: quanto leggi e scrivi questo indirizzo va là, mentre se si vuole fare esecuzione non si può o viceversa ---read-write only (RW)--- read-fetch only (XR))*

Il mapping avviene a **pagine** *(4096 byte o altre potenze di 2)* in modo tale da rendere le tabelle più piccole. Alcune zone sono mappate direttamente su disco (**Page file**) e non su memoria fisica → per tenere dei dati che il programma vuole effettivamente usare ma che sono troppo grossi per la RAM fisica. Dunque, il sistema operativo salva in memoria fisica, se non basta va nel disco *(quando uso le cose del disco, il sistema operativo vede se può mettere in memoria fisica o meno).*
Le zone grigie degli spazi virtuali sono inaccessibili, se il programma prova ad accedere lì, viene ucciso.



Tra processore e RAM è presente la **memoria cache** che serve a velocizzare il tutto con l'idea che io posso prevedere cosa mi serve: **PRINCIPIO DI LOCALITÀ**: *Se ho fatto accesso ad indirizzo I, farò anche accesso ad indirizzi I+Δ.*



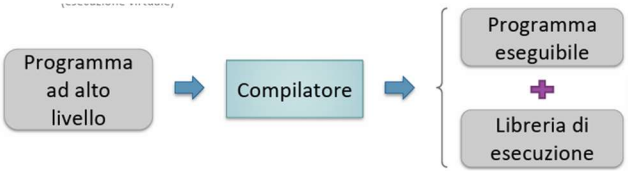
Memoria cache organizzata su **più livelli** con diversa capacità e velocità.
CPU possono avere vari core che leggono da memorie piccolissime ma velocissime(L1). Se serve riempire L1, i dati vengono presi da L2; se serve riempire L2: i dati vengono presi da L3; se serve riempire L3 i dati vengono presi dalla DRAM che contiene il sapere totale.

Concorrenza.:
Se i dati sono condivisi, devo invalidare la mia cache e andare a leggere dalla DRAM e per scriverli devo fare in modo che scendano fino alla DRAM.
***NOTA, se non si gestisse bene la concorrenza**: Se Core0 sta incrementando una variabile, questa variabile incrementata finisce in L1 del core0; se core3 vuole implementare quella stessa variabile, non riesce a vedere che l'altro l'ha incrementata e si rischia che entrambi facciano tale operazione*

***NOTA**: Ci sono vincoli e restrizioni di cui dobbiamo essere coscienti → es: garbage collector che elimina quando ha voglia lui la memoria.*

Modello di esecuzione di un programma: specifico del linguaggio di programmazione.

Il **compilatore trasforma** le istruzioni scritte nel linguaggio di alto livello in comportamenti tipici *in linguaggio del processore* (macchina). Inframezzate a queste istruzioni (traduzioni) vengono inserite delle *chiamate a libreria* (*RunTimeLibrary*) per fare in modo che l'astrazione funzioni.



Il tutto può essere a questo punto eseguito da una macchina fisica (CPU) o subire ulteriori trasformazioni.

Librerie di esecuzione:

Offrono meccanismi base per funzionamento e si interfacciano con il sistema operativo nascondendo le differenze.
Due tipi di **funzioni**:

- **Invisibili** al programmatore, inserite in fase di compilazione per supporto esecuzione *(controllo, stack…)*
- Funzionalità **standard**, gestendo strutture dati ausiliarie e/o richiedendo al S.O. funzioni particolari *(malloc, fopen)*

***NOTA**: In C e C++ i programmi pensano di essere gli unici in esecuzioni e quindi di poter andare in memoria in qualsiasi posto tra 0 e 2^{N-1}, anche se non è vero. → grazie al meccanismo di indirizzamento virtuale.*

NOTA: programma inizia da una certa cella ± Δ dove Δ è random → in modo tale da rendere più difficile la vita ai virus.

FLUSSO ESECUZIONE

In C immagino che le cose vadano esattamente nell'ordine in cui le ho scritte anche se non è detto che succeda questo dato che il compilatore può decidere che certe cose le può scambiare finché nessuno se ne accorge; il processore può fare cose diverse finché nessuno se ne accorge.

All'interno del flusso principale di esecuzione, possono essere abilitati flussi di esecuzione secondari →**THREAD**: mi permettono di fare delle cose in contemporanea abilitando esecuzione concorrente.

ORDINE IN CUI AVVENGONO LE COSE

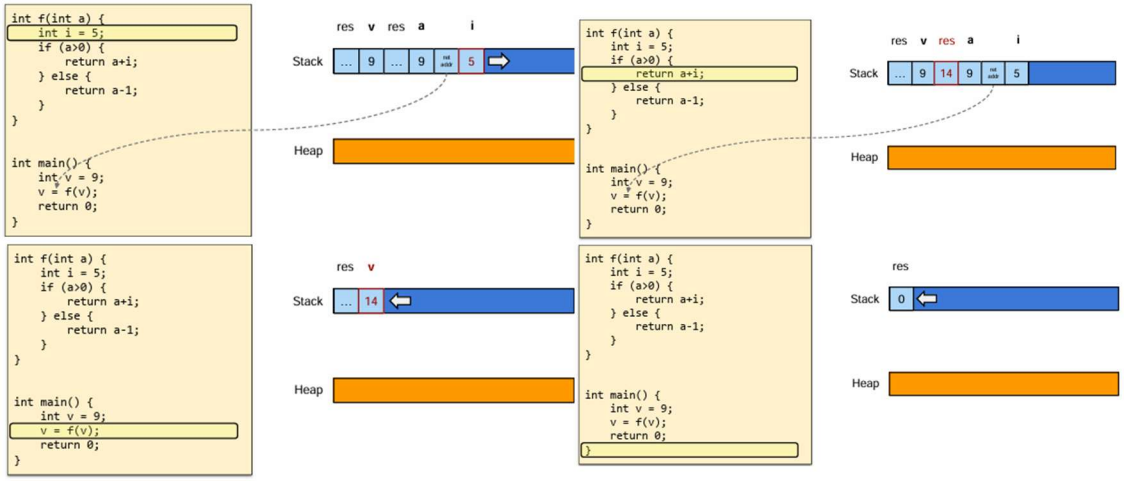
- Setup sistema operativo
- Costruttore delle variabili globali
- Main
- Distruttore variabili globali

Strutture:

Stack

Struttura a pila che permette di gestire le chiamate tra funzioni, i punti di ritorno e le variabili locali etc.

- Viene **allocato automaticamente** all'avvio di un programma
  - Parte da un estremo
  - Si estende verso il basso
- Mentre leggiamo un programma lo stack si allarga e si contrae
- Dimensione **finita** → *stack overflow*
  - Limita profondità ricorsione
  - Limita variabili locali



***NOTA**: i dati che sono nello stack durano al massimo quanto il blocco {} in cui sono definiti (infatti il valore di ritorno è allocato dal chiamante e non dal chiamato -altrimenti non arriverebbe-)*

Ad esempio:

- *se ho una variabile locale in un if, viene creata ad inizio if ed eliminata alla chiusa graffa.*
- *se ho una variabile locale in un for(loop 10), la variabile locale sarà creata e distrutta 10 volte.*

Heap

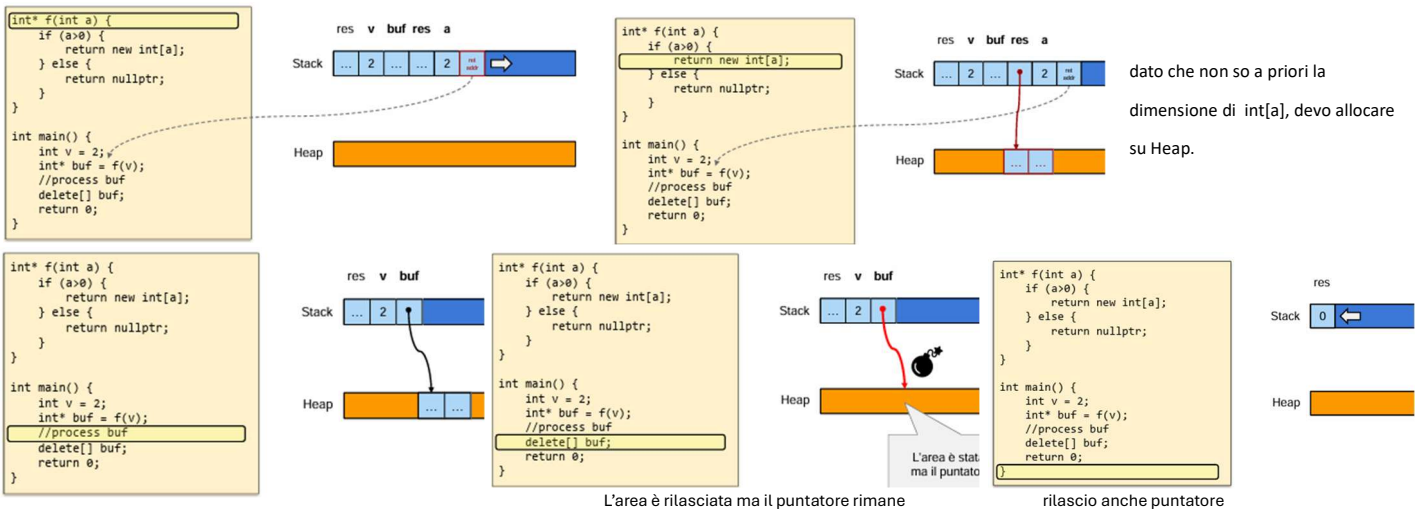
serve a gestire la memoria dinamica → mi dà blocchi di memoria disponibili finché mi servono.

Tutte le volte che un dato ha un **ciclo di vita non collegato a quello della funzione** in cui nasce oppure è troppo **grosso** oppure ha una **dimensione non nota** al momento della compilazione → metto in Heap e non in stack.

Le zone nell'Heap non hanno un nome (*a differenza di quanto avveniva nello stack dove le aree hanno un nome),* sono solo **accessibili tramite puntatori** → devo memorizzare in una variabile di tipo puntatore (*tipo* var*) e per accedere devo deferenziare (**var*).

Nonostante sia di dimensione elevata, dopo un po' anche lo Heap si esaurisce e quindi se gli chiedo un pezzo, il programma si rompe → Per evitare ciò, **quando la zona non serve più, va liberata.**

È compito del programmatore di liberare Heap.



dato che non so a priori la dimensione di int[a], devo allocare su Heap.

L'area è rilasciata ma il puntatore rimane

rilascio anche puntatore

ORGANIZZAZIONE DELLO SPAZIO DI INDIRIZZAMENTO:

- **Codice eseguibile**
  - Istruzioni in codice macchina
  - Accesso in lettura & esecuzione
- **Costanti**
  - Acceso in lettura
- **Variabili globali**
  - Lettura/Scrittura
- **Stack**
  - Indirizzi e valori di ritorno, parametri, variabili locali
  - Lettura/Scrittura
- **Heap**
  - Insieme di blocchi in memoria disponibili per allocazione dinamica
  - Gestiti tramite funzioni di libreria che li frammentano e ricompattano in base alle richieste del programma

- Nel sistema operativo Linux, è facile verificare come sia organizzato lo spazio di indirizzamento di un programma in esecuzione (processo)
  - Ogni processo è identificato univocamente da un numero intero (PID)
  - Il comando **ps -ef** permette di elencare tutti i processi esistenti all'atto della sua esecuzione
- Il sistema operativo crea, per ciascun processo attivo, un file virtuale denominato **/proc/<pid>/maps** che descrive lo spazio di indirizzamento con informazioni sui diversi segmenti al suo interno

```
$ cat /proc/4742/maps
…altre righe…
7f4e3161d000-7f4e3161e000 r-xp 00001000 00:00 3538         /home/user/testmem
7fffc62c6000-7fffc62e7000 rw-p 00000000 00:00 0            [heap]
7fffcdb6d000-7fffce36d000 rw-p 00000000 00:00 0            [stack]
7fffceaee000-7fffceaef000 r-xp 00000000 00:00 0            [vdso]
```

Queste due strutture crescono e si contraggono mentre il programma va in esecuzione. 5 MARZO

Variabili globali:

- **indirizzo fisso** determinato dal compilatore
- accessibili in ogni momento
- costruttore delle variabili globali parte prima che inizi il main, *con un eventuale valore di inizializzazione*

Variabili locali:

- hanno un **indirizzo relativo** alla cima dello stack *(es: BasePointer+82)*
- ciclo di vita coincidente con quello della funzione in cui sono dichiarate
- hanno un valore iniziale casuale, contenuto non prevedibile

Variabili dinamiche:

- contengono un dato ed è possibile referenziarle solo con **puntatore**
  - inizia ad esistere quando la alloco (malloc);
  - cessa di esistere quando la rilascio (free).
- indirizzo assoluto che si può sapere solo in runtime
- può essere inizializzata, dipende dal costruttore
- necessitano di un sistema di supporto per capire dove salvare nell'Heap questa variabile
  - fornito dalla libreria di esecuzione del S.O.

  - `void *malloc(size_t s)`
  - `void *calloc(int n, size_t s)`
  - `void *realloc(void* p, size_t s)`

***NOTA**: allocazione e rilascio sono operazioni lente e complesse che consumano*

allocazione:

- chiamo il **costruttore**: definisce cosa fare
  - chiamato esplicitamente quando si dichiara una variabile

- Per allocare sequenze di oggetti, C++ offre il costrutto `new NomeClasse[numero_elementi]`
  - Si indica il numero di oggetti consecutivi da allocare tra le quadre
  - Inizializza i singoli oggetti con il costruttore di *default*
  - Restituisce il puntatore all'inizio dell'*array*

In C++ viene definito il costrutto

`new NomeTipo{argomenti…}`

- Alloca nello heap un blocco di dimensioni opportune
- Invoca il costruttore della classe sul blocco per inizializzare il suo contenuto
- Restituisce il puntatore all'oggetto inizializzato

Dal C++v11 ne esistono 2 versioni:

- `new NomeTipo{args…}`
- `new (std::nothrow) NomeTipo{args…}`
- La prima versione genera un'eccezione invece di ritornare un puntatore non valido (**nullptr**) se non è possibile trovare un'area grande a sufficienza per contenere il tipo di dato richiesto

rilascio: restituisce il pezzo di memoria che ha ricevuto

- chiamo il **distruttore**: definisce cosa fare in fase di rilascio *(es: chiudere il file)*
  - *chiamato implicitamente dal compilatore quando si effettua una delete*

un blocco sia rilasciato dalla **funzione duale** di quella con cui è stato allocato

- `malloc(…) / free(…), new / delete, new …[num_elements] / delete[ ]`

- nel caso in cui si effettua una doppia delete
  - se c'è un'implementazione lenta ma attenta → *rigetta*
  - se implementazione veloce → *prende per buono che l'indirizzo che gli passo sia in uso e aggiorna le strutture dati spaccando tutto.*
    - *Se rilascio una cosa sbagliata → casino totale*
    - *se non rilascio → riempio tutto lo spazio*

Puntatori

Sono estremamente potenti ma anche pericolosi. → possono causare undefined behaviour.

USO:

- dare accesso alla variabile senza doverla copiare
- allocati per uno scopo particolare
- possono essere **resi invalidi** se valore 0/NULL/nullptr
- **PROBLEMATICHE**
  - Se il valore è diverso da 0 non posso sapere se è effettivamente valido
  - non posso sapere quanto è grosso il blocco puntato
  - non so fino a quando è garantito l'accesso → scadenza
  - non so se posso modificare ancora (attraverso i cast potrei aver cambiato da const a char)
  - non so se devo rilasciare io
    - nel caso in cui sia responsabile del rilascio, devo sapere anche quando devo rilasciarlo
  - uso per dare accesso al dato oppure per opzionalità?

- Tale blocco può appartenere ad altri oggetti
  - `int A=10;`
    `int* pA = &A;`
- Essere allocato allo scopo
  - `int* pB = new int{24};`

Dunque, **in C/C++ i puntatori sono usati per accedere "qui ed ora"** ad un'informazione contenuta in un'altra struttura dati **oppure** sono anche usati per indicare ad una **funzione dove deve depositare i risultati**

- *responsabilità e gestione non sono dell'osservatore*
- *se in sola lettura, antepongo const*

```
{
  const char* ptr = "Quel ramo del lago di Como...";

  //conta gli spazi
  int n=0;

  //usa l'aritmetica dei puntatori
  for (int i=0; *(ptr+i)!=0; i++) {

    //usa il puntatore come fosse un array
    if ( isSpace(ptr[i]) ) n++;
  }
  //altro...
}
```

Sono usati per **accedere ad un blocco di dati**:

- il problema è che si passa il puntatore al primo e non il nr di elementi

(in C++ hanno introdotto anche la dimensione del vettore)

```
int* read_data() {
  unsigned int size = count_available_data();
  if (size > 0 ) {
    int* ptr= new int[size];
    consume_data(ptr, size);
    //indica operazione eseguita
    //correttamente
    return ptr;
  } else
    //nessun dato disponibile
    return nullptr;
}

int* result = read_data();
...
if (result) delete[] result;
```

Sono usati per accedere **ai dati dinamici**

- sicuramente responsabilità di rilascio
  - devo essere certo che qualcuno rilasci
  - C++ aiuta in questo in quanto ha costruttore e distruttore

```
bool read_data1(int* result) {

  //Se il puntatore sembra valido
  //e ci sono dati...
  if (result!=nullptr && some_data_available() )

    //accedi in scrittura all'indirizzo indicato dal chiamante
    *result = get_some_data();

    //indica operazione eseguita correttamente
    return true;
  } else
    //operazione fallita
    return false;
}
```

Sono usati per **opzionalità** di risultato

- puntatore valido se ho un ritorno
- 0 se non ritorno

Sono usati per **liste, grafi, mappe**

```
struct simple_list {
  int data;
  struct simple_list *next;
};

struct simple_list *head;
// head è responsabile di tutte le proprie parti
// quando si rilascia la lista, occorre liberarne
// tutti gli elementi
```

In C++ c'è un concetto di movimento con il quale perdi il possesso, cedendolo all'altro; mentre in C la gestione della memoria è totalmente affidata al programmatore.

RESP DEL PROGRAMMATORE:

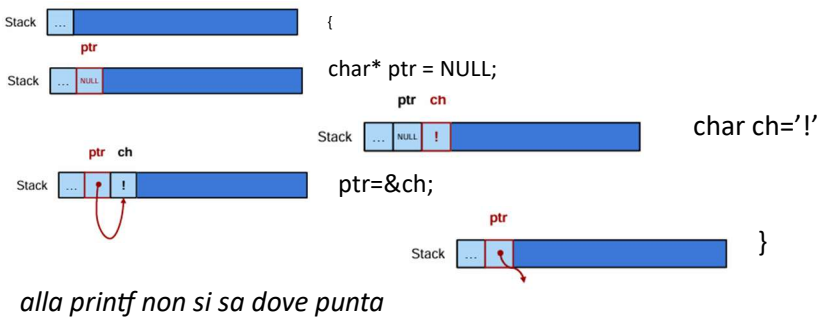
- Limitare accessi ad un blocco (spazio/tempo)
- Non assegnare a puntatori valori che corrispondono ad indirizzi non mappati
- Rilasciare tutta la memoria dinamica allocata

RISCHI

- Accedere ad un indirizzo quando il ciclo di vita è terminato può avere effetti impredicibili → **Dangling pointer**
  - *La memoria indirizzata può essere inutilizzata o in uso ad altre parti del programma*
- Se non rilascio tutta la memoria che ho allocato → **memory leakage** → spreco risorse
  - *Rischio di finire tutto lo spazio*
- Rilascio multiplo della memoria → Double free
  - Se rilascio la memoria più volte corrompe le strutture usate dallo heap → **double free**
- Se non si inizializza un puntatore e lo si usa → wild pointer
- Se so assegna un puntatore ad un indirizzo non mappato nello spazio di indirizzamento
  - *Interruzione del processore → S.O. intercetta interruzione e termina processo*

DANGLING POINTER → heap vuoto

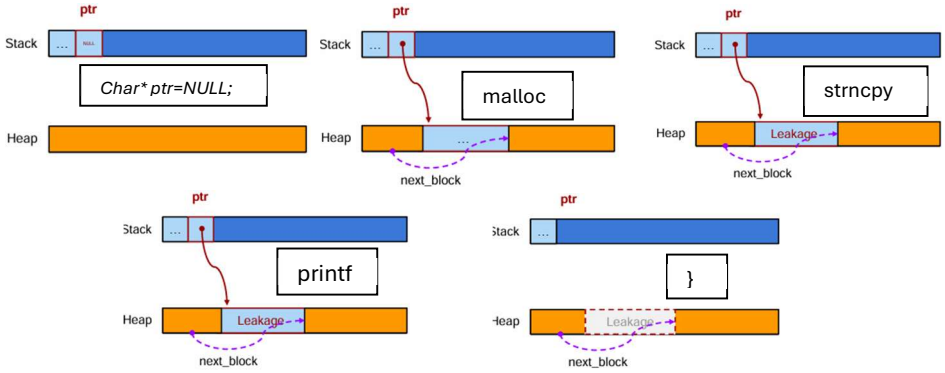
```
{
  char* ptr = NULL;

  {
    char ch='!';
    ptr = &ch;

  }

  printf("%c", *ptr);
}
```



alla printf non si sa dove punta

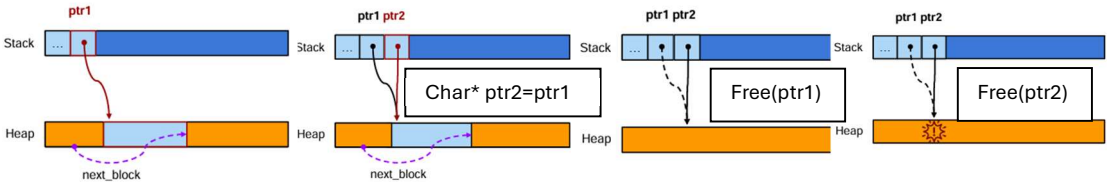
MEMORY LEAKAGE

```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);

  strncpy(ptr,10,"Leakage!");

  printf("%s\n", ptr);
}
```



DOUBLE FREE

```
{
  char* ptr1 = malloc(10);
  char* ptr2 = ptr1;
  // use ptr1 and/or ptr2
  free(ptr1);
  free(ptr2);
}
```



Gestione della memoria:

Chi alloca un blocco di memoria è **responsabile** di mettere in atto un meccanismo che garantisca il successivo **rilascio** → possessore del blocco

POSSESSO: chi ha il possesso, ha il diritto di accedere al blocco ma anche il dovere di rilasciarlo.
Nel caso in cui ci siano i puntatori che posseggano il blocco, devo decidere chi tra i due lo rilascia. Inoltre, dato un indirizzo non nullo, non è possibile distinguere se sia valido, a quale area appartenga, se bisogna rilasciare.
(es: su java quando l'ultimo che ha il possesso non è più interessato, allora è rilasciabile)
NOTA: puntatore di tipo WEAK, partecipano ma non posseggono

DIPENDENZE: strutture dati complesse che si appoggiano su blocchi di memoria ausiliari in cui memorizzare le informazioni che gestiscono → es: vettori di dati dinamici con tipo generico

Possono esistere diverse dipendenze: legate alla memoria, legati al S.O: (apertura/chiusura file)

A differenza di altri linguaggi, RUST è memory safe e non fa garbage collector.