Allocazione della memoria

Programmazione di Sistema

A.A. 2017-18

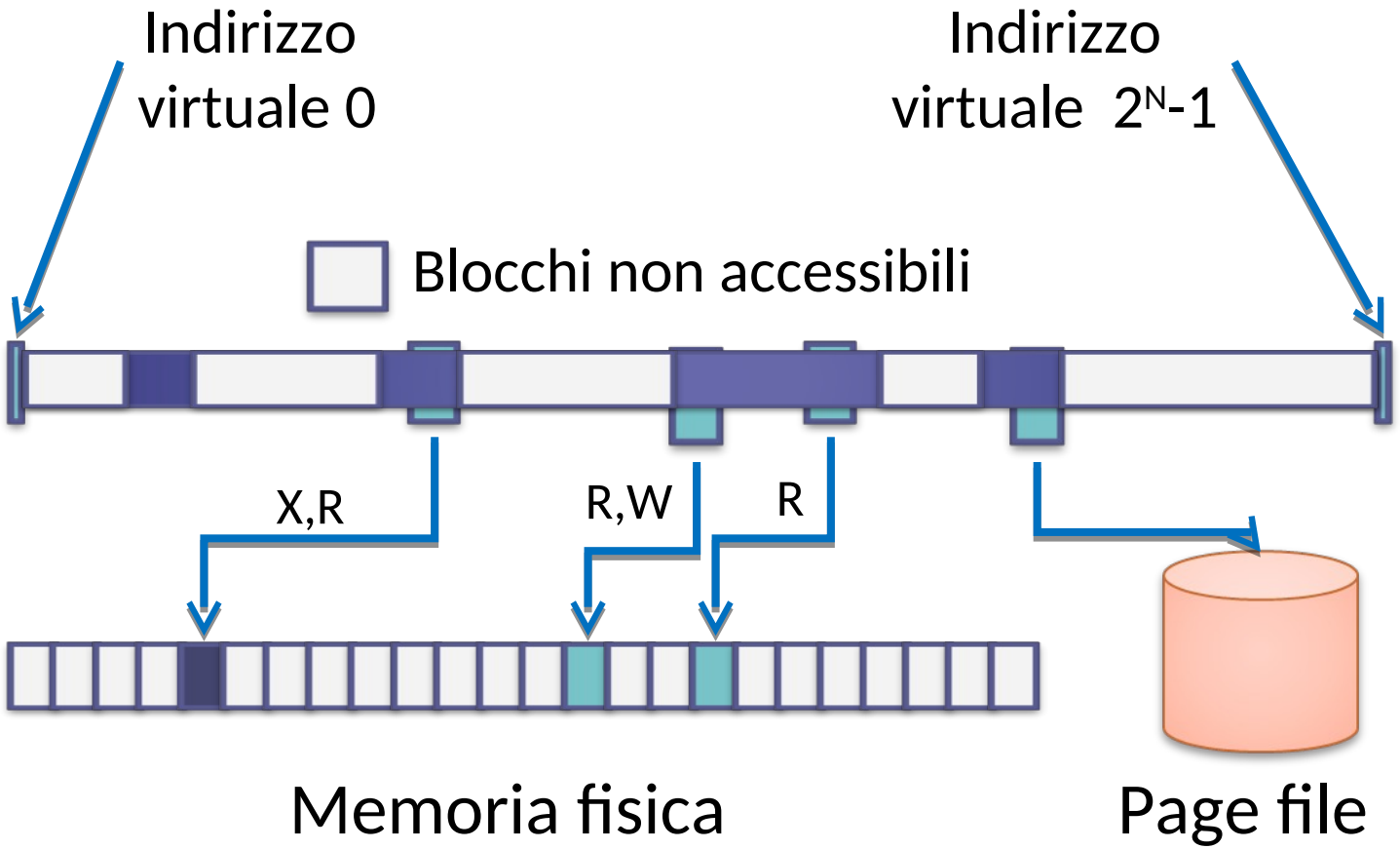
Argomenti

- Allocazione statica e dinamica
- Puntatori e loro utilizzo
- Allocazione in Linux
- Allocazione in Windows

Spazio di indirizzamento

- L’esecuzione di un programma avviene nel suo spazio di indirizzamento
  - Insieme di locazioni di memoria accessibili tramite indirizzo virtuale
  - Sottoinsieme delle celle indirizzabili, gestito dal sistema operativo

Spazio di indirizzamento



Classi di allocazione

- Il sistema operativo/la libreria di esecuzione offrono aree diverse
  - In funzione dell'utilizzo e del ciclo di vita
  - Leggibili, scrivibili, eseguibili

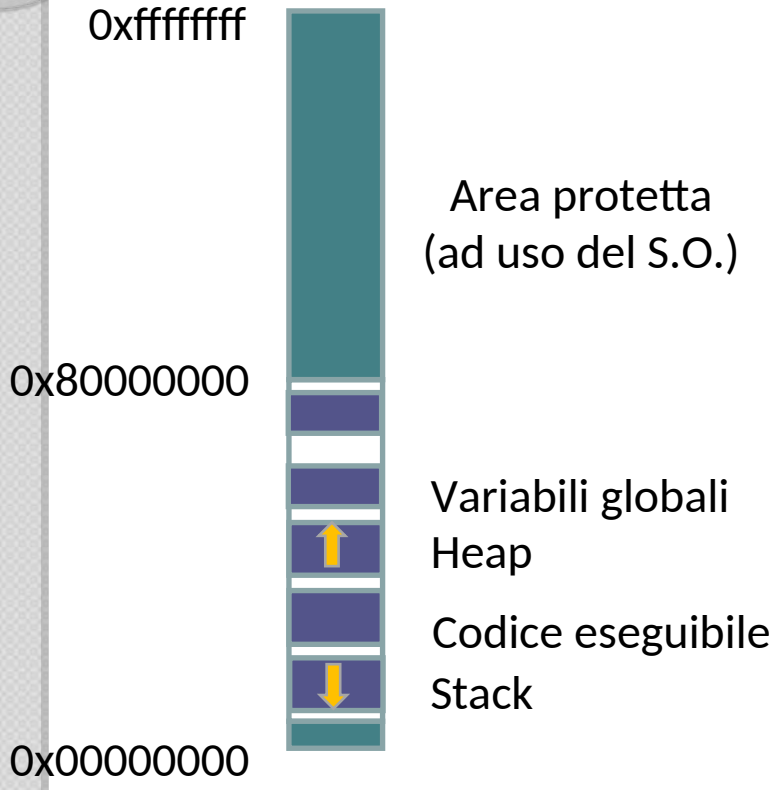
Criteri di accesso

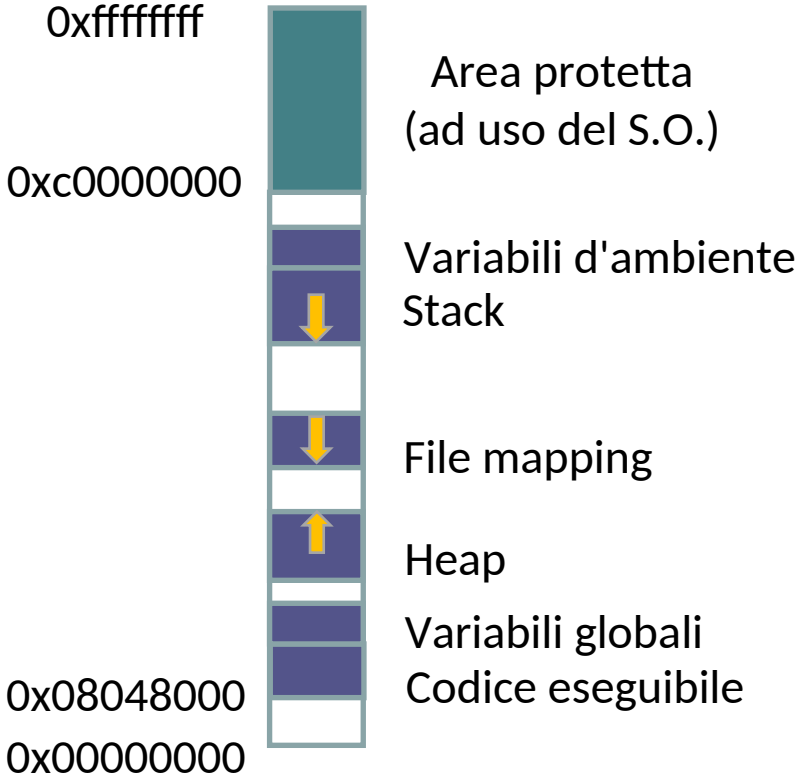
- La corrispondenza tra indirizzi virtuali e pagine fisiche è corredata di metadati
  - Definiscono quali operazioni sono lecite sulla memoria
  - eXecute, Read, Write, Copy_on_write
- Accessi a locazioni non mappate o in violazione dei criteri indicati comportano l'interruzione della CPU
  - Il processo viene terminato
  - Access violation (Windows) o segmentation fault (Linux)

Uso della memoria

- Quando un processo viene creato, il suo spazio di indirizzamento viene popolato con diverse aree
  - Ciascuna dotata di propri criteri di accesso
  - Funzionali a supportare i modelli di esecuzione dei linguaggi C/C++

Spazio di indirizzamento in Windows e Linux





Organizzazione dello spazio di indirizzamento

- Codice eseguibile
  - Contiene le istruzioni in codice macchina
  - Accesso in lettura ed esecuzione
- Costanti
  - Accesso in sola lettura
- Variabili globali
  - Accesso lettura/scrittura
- Stack
  - Contiene indirizzi e valori di ritorno, parametri e variabili locali
  - Accesso lettura/scrittura

Organizzazione dello spazio di indirizzamento

- Free store o heap
  - Insieme di blocchi di memoria disponibili per l'allocazione dinamica
  - Gestiti tramite funzioni di libreria (malloc, new,
    free, …) che li frammentano
    e ricompattano in base alle richieste del programma

Ciclo di vita delle variabili

- Il modello di esecuzione del C distingue diverse classi di variabili
  - Globali, locali, dinamiche
- Ciascuna classe ha un proprio ciclo di vita
  - Intervallo di tempo in cui è garantito l'accesso alle informazioni

Variabili globali e locali

- Le variabili globali hanno un indirizzo fisso, determinato dal compilatore e dal linker
  - Accessibili in ogni momento
  - All'avvio del programma, contengono l'eventuale valore di inizializzazione
- Le variabili locali hanno un indirizzo relativo alla cima dello stack
  - Ciclo di vita coincidente con quello della funzione/blocco in cui sono dichiarate
  - Valore iniziale casuale

Variabili dinamiche

- Hanno un indirizzo assoluto, determinato in fase di esecuzione
  - Accessibili solo tramite puntatori
  - Il programmatore ne controlla il ciclo di vita
  - Il valore iniziale può essere inizializzato o meno
- L'uso di questo tipo variabili presuppone un'infrastruttura di supporto che offra meccanismi di allocazione e di rilascio
  - Fornita dalla libreria di esecuzione e dal S.O.

Allocazione della memoria

- La libreria standard C offre vari meccanismi per ottenere un puntatore dinamico
  - void *malloc(size_t s)
  - void *calloc(int n,size_t s)
  - void *realloc(void* p,size_t s)
- In C++ viene definito il costrutto new NomeClasse
  - Alloca nello heap un blocco di dimensioni opportune
  - Invoca il costruttore della classe sul blocco
  - Restituisce il puntatore all'oggetto inizializzato

Allocazione della memoria

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

Rilascio della memoria

- Opportune funzioni di libreria permettono di restituire i blocchi precedentemente allocati
  - Organizzandoli in una lista in base alla dimensione e altri criteri
- Poiché ogni funzione di allocazione mantiene le proprie strutture dati
  - Occorre che un blocco sia rilasciato dalla funzione duale di quella con cui è stato allocato
  - malloc/free, new/delete, new[]/delete[]
- Se il blocco viene rilasciato con la funzione sbagliata
  - si rischia di corrompere le strutture dati degli allocatori, con conseguenze imprevedibili

Puntatori

- Permettono l'accesso diretto ad un blocco di memoria
  - Appartenente ad altri oggetti
    - int A=1O; int* pA = &A;
  - Allocato allo scopo
    - int* pB = new int(24);
- Possono essere invalidi
  - Il valore O
  - La macro NULL ((void *)O)
  - La parola-chiave nullptr (C++11)
- Quando si usa un puntatore, occorre stabilire quali responsabilità/permessi sono associati al loro uso

Le ambiguità dei puntatori

- Quanto è grosso il blocco puntato?
- Fino a quando è garantito l'accesso?
- Se ne può modificare il contenuto?
- Occorre rilasciarlo?
- Lo si può rilasciare o il blocco è accessibile tramite una copia del puntatore?

Usi dei puntatori

- Come strumento per accedere qui ed ora ad un'informazione fornita da altri
  - La responsabilità per la gestione della memoria del dato è totalmente esterna all'osservatore
- Caso più semplice e alquanto frequente

Puntatori

```
int read_data1(int* result) {

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

    //accedi in scrittura alla memoria
    *result = get_some_data();

    //indica operazione eseguita correttamente
    return 1;
  } else
    //operazione fallita
    return 0;
}
```


Puntatori

- Per accedere ad array monodimensionali di dati
  - Il compilatore trasforma gli accessi agli array in operazioni aritmetiche sui puntatori
  - Si perde di vista l'effettiva dimensione della struttura dati
- Occorre fare attenzione a non spostare il puntatore al di fuori dell'effettiva zona di sua pertinenza

Puntatori

```
{

  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++;
  }
  ...
}
```


Puntatori

- Come modo per accedere ad un dato memorizzato sullo heap
  - Il ciclo di vita del dato è slegato da quello del blocco di codice che lo ha allocato
  - Chi è responsabile del suo rilascio e quando va fatto?
- E' il caso base di strutture dati dinamiche

Puntatori

```
int* read_data2() {
  if (some_data_available() ) {
    int* ptr= (int*) malloc( sizeof(int) );
    *ptr = get_some_data();
    //indica operazione eseguita correttamente
    return ptr;
  } else
    //operazione fallita
    return NULL;
}

int* result = read_data2();
...
free(result);
```


Puntatori

- Strumento per implementare strutture dati composte
  - Liste, grafi, mappe, ...
  - La struttura nel suo complesso è responsabile della gestione di tutte le sue parti

```
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
```


Problemi legati ai puntatori

- In C, la gestione della memoria è totalmente affidata al programmatore
  - Non ci sono supporti sintattici per identificare l'uso associato ad un puntatore

Responsabilità del programmatore

- Limitare gli accessi ad un blocco
  - Nello spazio
    - Non accedere alle locazioni che lo precedono o che lo seguono
  - Nel tempo
    - Non accedere al blocco al di fuori del suo ciclo di vita
- Non assegnare a puntatori valori che corrispondono ad indirizzi non mappati
  - Possibile nel caso di cast o di assegnazione improprie
- Rilasciare tutta la memoria dinamica allocata
  - Una e una sola volta

Rischi

- Accedere ad un indirizzo quando il corrispondente ciclo di vita è terminato, ha effetti impredicibili
  - *Dangling pointer*
  - La memoria indirizzata può essere inutilizzata, in uso ad altre parti del programma o non mappata
- Non rilasciare la memoria non più in uso, spreca risorse del sistema
  - *Memory leakage*
  - Se si continua ad allocare senza mai rilasciare, si può saturare lo spazio di indirizzamento

Rischi

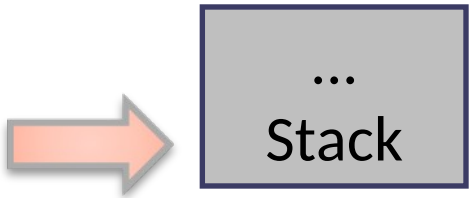
- Se si assegna ad un puntatore un indirizzo non mappato e si usa il puntatore, viene generata un'interruzione
  - Il S.O., per lo più, termina il processo
- Se non si inizializza un puntatore e lo si usa, il suo contenuto potrebbe puntare ovunque
  - *Wild pointer*
  - Causando una violazione d'accesso
  - Oppure corrompendo un'area di memoria in uso ad altre parti del programma

Dangling Pointer

```
{
  char* ptr = NULL;

  {  // inizio di un nuovo blocco
    char ch='!';
    ptr = &ch;

  }   // fine blocco: lo stack si contrae
      // le variabili qui definite cessano di esistere

  printf("%c", *ptr);    //contenuto impredicibile

}
```

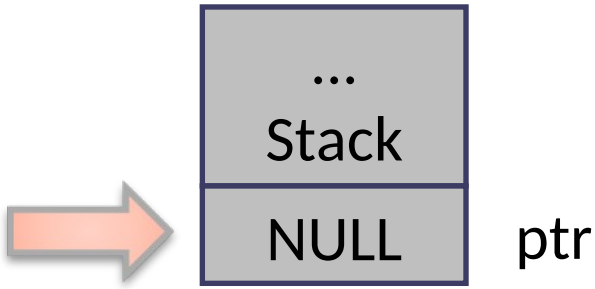

Dangling Pointer

```
{
  char* ptr = NULL;

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

  }

  printf("%c", *ptr);

}
```



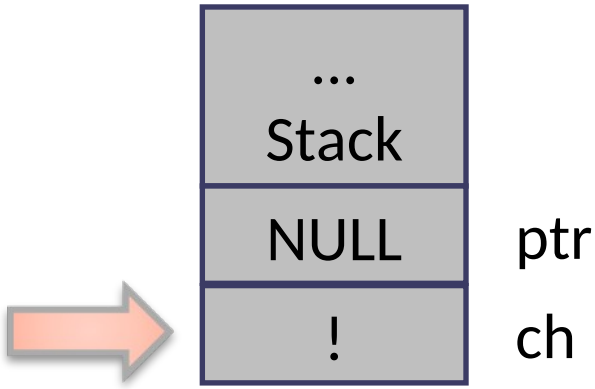
Dangling Pointer

```
{
  char* ptr = NULL;

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

   }

   printf("%c", *ptr);

}
```



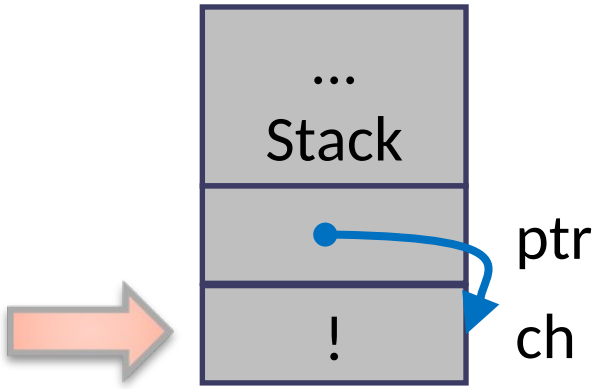
Dangling Pointer

```
{
  char* ptr = NULL;

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

  }

  printf("%c", *ptr);

}
```



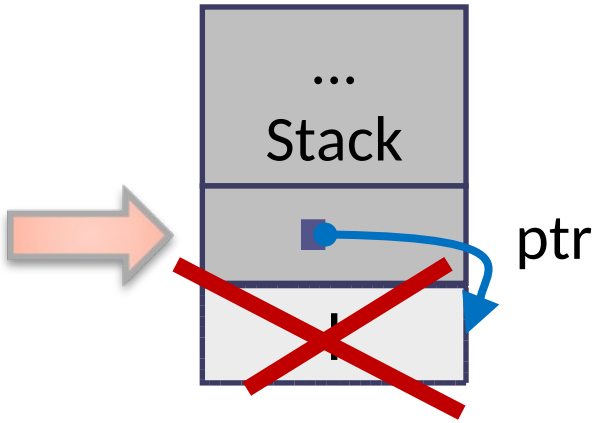
Dangling Pointer

```
{
  char* ptr = NULL;

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

  }

  printf("%c", *ptr);

}
```



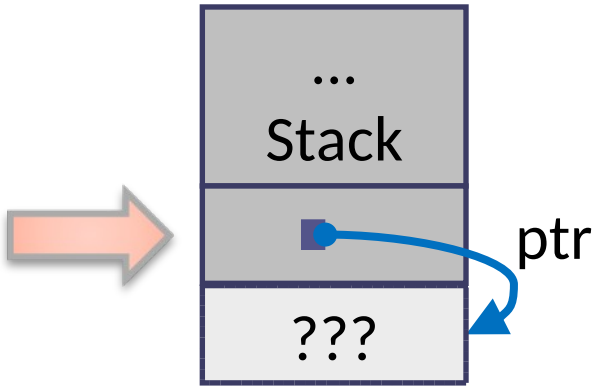
Dangling Pointer

```
{
  char* ptr = NULL;

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

  }

  printf("%c", *ptr);

}
```



Dangling Pointer

```
{
  char* ptr = NULL;

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

  }

  printf("%c", *ptr);

}
```

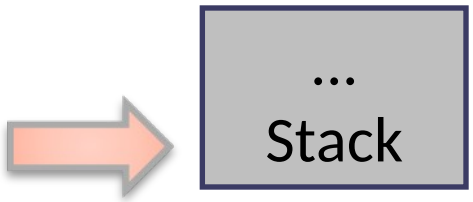


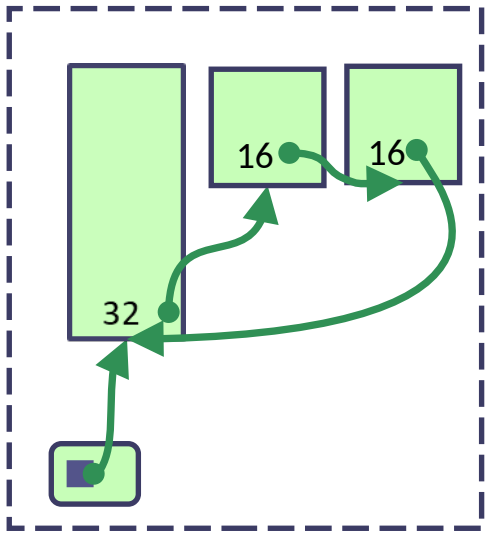
Memory leakage

```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);   //Alloco un blocco

  strncpy(ptr,10,"Leakage!"); //Lo uso

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

}                                          //Ne perdo le tracce
```


Memory leakage

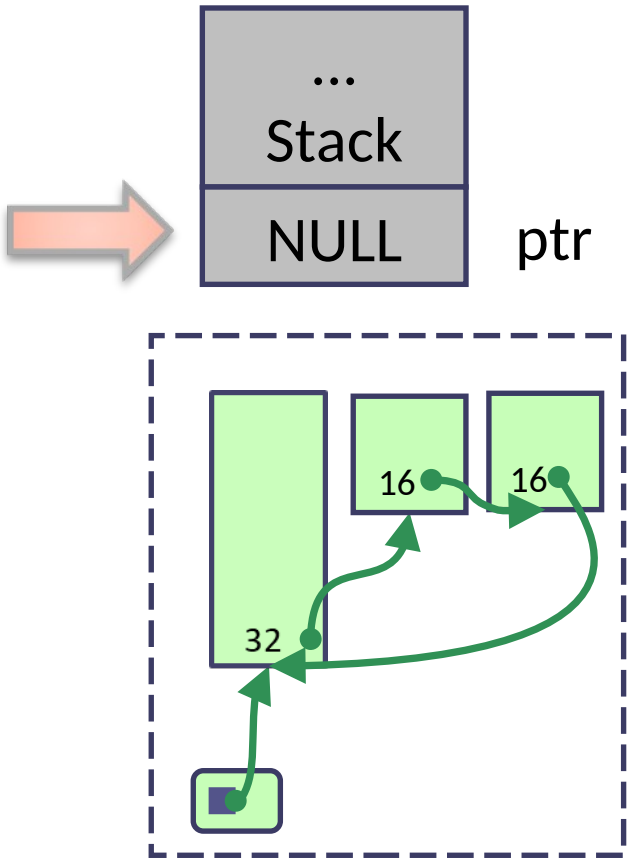
```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);

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

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

}
```





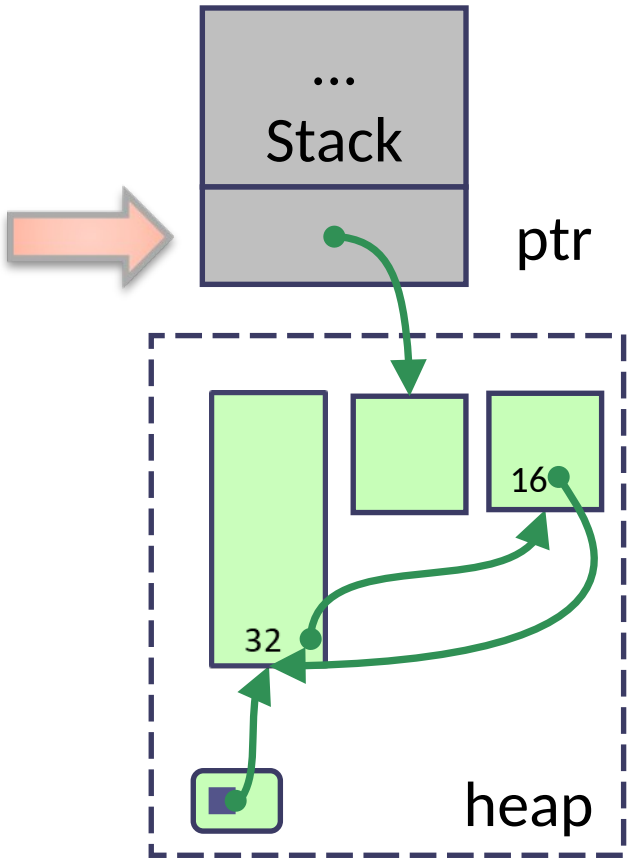
Memory leakage

```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);

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

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

}
```



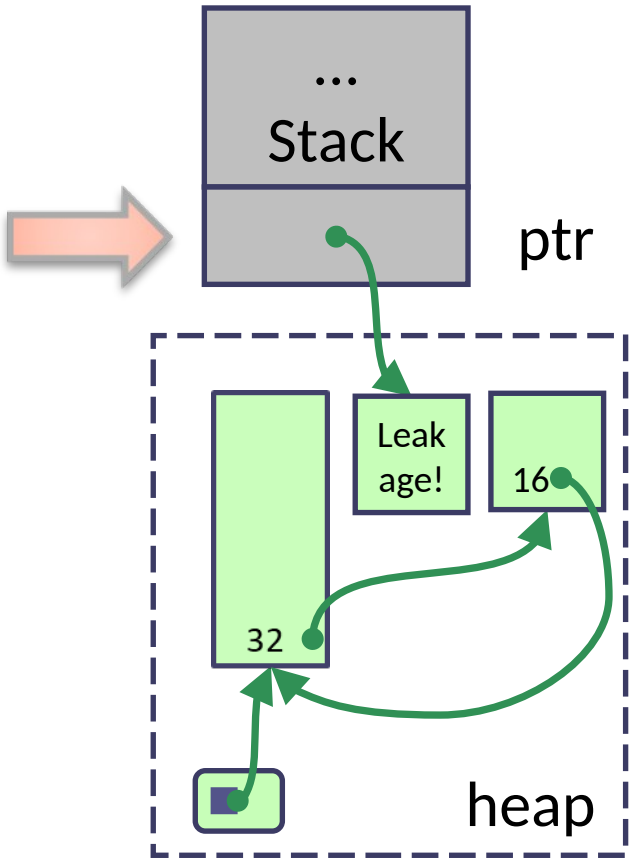
Memory leakage

```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);

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

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

}
```



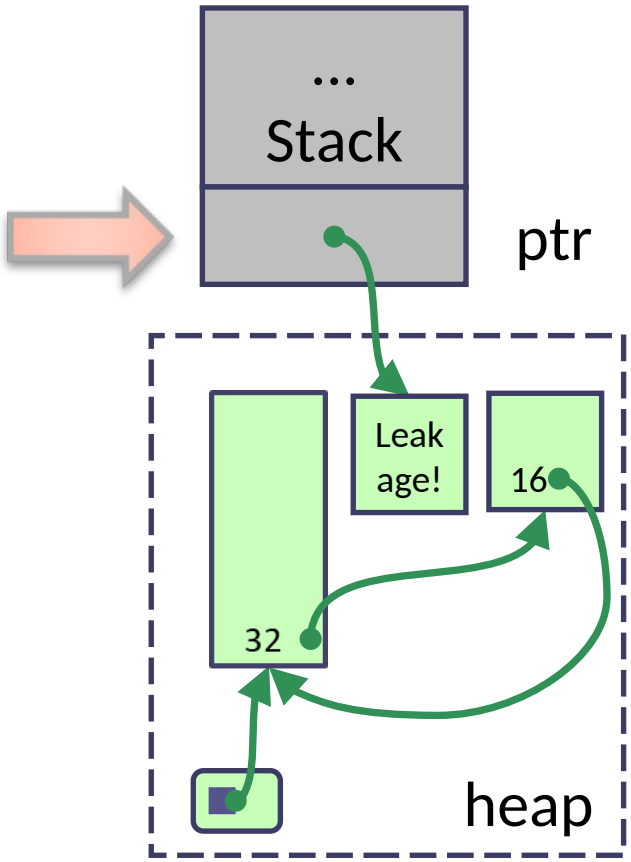
Memory leakage

```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);

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

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

}
```



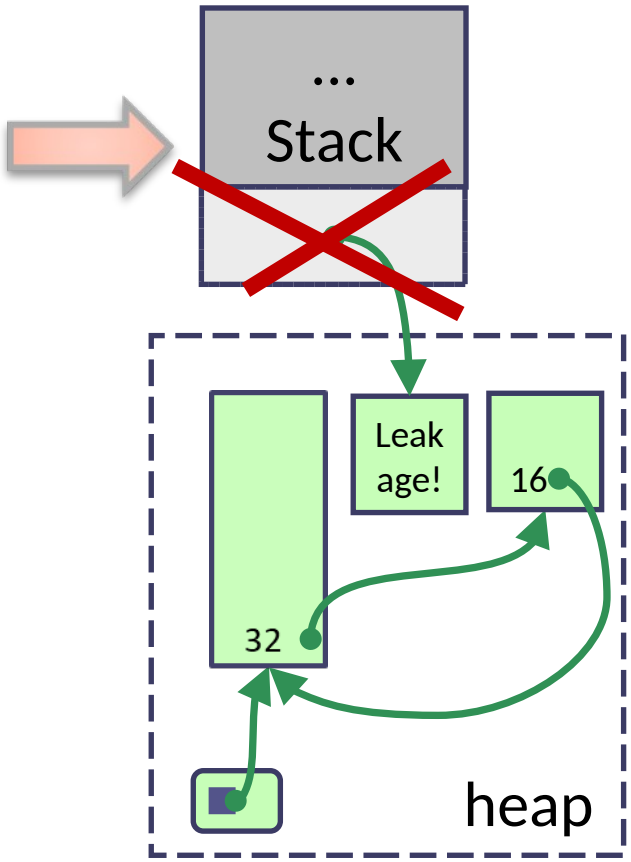
Memory leakage

```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);

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

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

}
```



Memory leakage

```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);

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

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

}
```



Gestire i puntatori

- Chi alloca un blocco di memoria è responsabile di mettere in atto un meccanismo che ne garantisca il successivo rilascio
  - Viene detto “possessore” del puntatore
- Cosa capita se un blocco di codice copia un puntatore?
  - Si trova coinvolto, suo malgrado, nel ciclo di vita
  - Occorre introdurre un meccanismo per gestire efficacemente la semantica di un puntatore

Possesso della memoria

- Il vincolo di rilascio risulta problematico per via di un ambiguità del linguaggio
  - Dato un indirizzo non nullo, non è possibile distinguere a quale area appartenga né se sia valido o meno
- Ogni volta in cui si alloca un blocco dinamico e se ne salva l'indirizzo in una variabile puntatore
  - Quella variabile diventa proprietaria del blocco
  - Ha la responsabilità di liberarlo

Proprietà della memoria

- Non tutti i puntatori posseggono il blocco a cui puntano
  - Se ad un puntatore viene assegnato l'indirizzo di un'altra variabile, la proprietà è della libreria di esecuzione
- Il problema si complica se un puntatore che possiede il proprio blocco viene copiato
  - Quale delle due copie è responsabile del rilascio

Tecniche di sopravvivenza

- Utilizzo di strumenti per la diagnosi dei processi
  - Valgrind (linux)
  - Dr.Memory (windows)
- Incapsulamento dei puntatori in apposite strutture dati
  - SmartPointer in C++

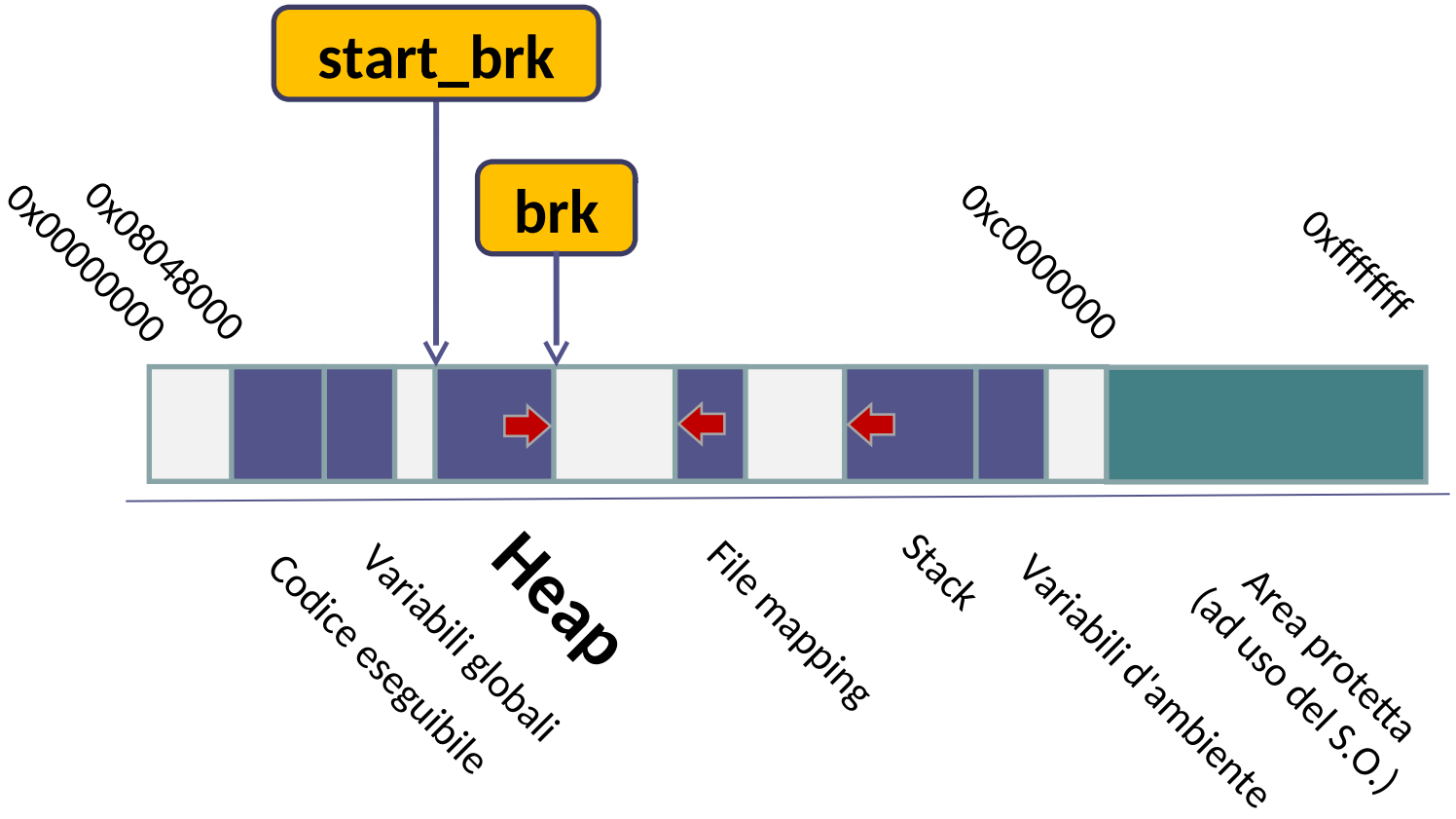
Spunti di riflessione

- Si crei un programma C che allochi e non rilasci un blocco di memoria
- Si cerchi di scoprire tale perdita utilizzando i programmi valgrind e Dr.Memory

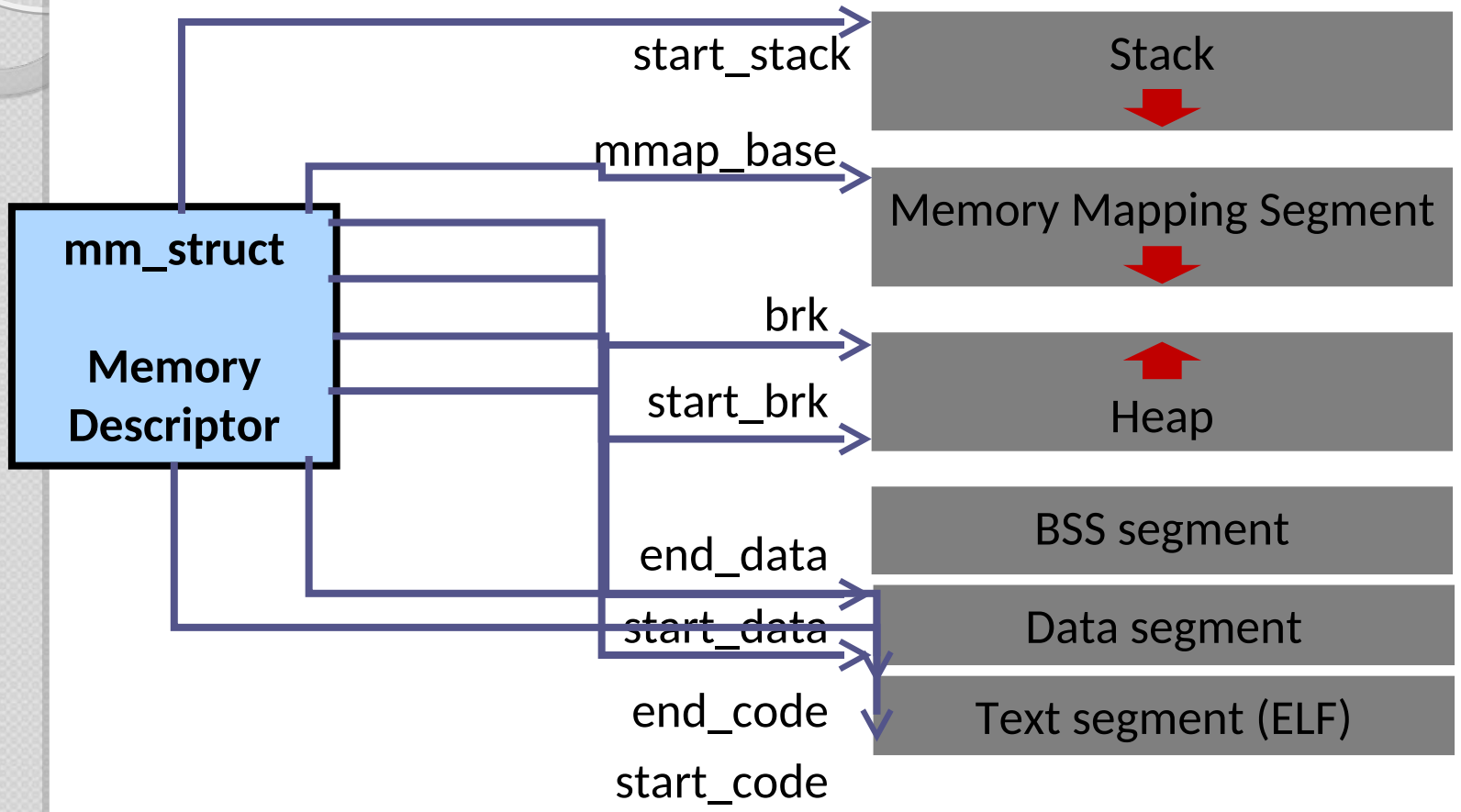
Allocazione in Linux

- Quando il processo viene inizializzato, viene creato un blocco di memoria a disposizione dello heap
  - Le funzioni di allocazione cercheranno di utilizzare la memoria disponibile per soddisfare le richieste
- Se nessuno dei blocchi liberi ha dimensioni sufficienti, occorre richiedere al S.O. altra memoria
  - I dettagli di tale operazione dipendono dall'implementazione

Lo spazio di indirizzamento in Linux



La rappresentazione interna



Allocare altro spazio

- Il sistema mantiene compatta l'area dello heap
  - Delimitandola con due puntatori interni al sistema
    - start_brk – indica l'inizio dello heap

```
int brk(void *end_data_segment);
```

- Fissa la dimensione del segmento dati specificando il valore limite, chiamato program break
  - La memoria disponibile varia di conseguenza
  - In caso di errore ritorna -1

La system call sbrk()

- Sposta la locazione del program break della quantità indicata
  - Incrementando o diminuendo l'area dello heap
  - In caso di errore, ritorna (void*)-1

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


Politiche di allocazione

- Per evitare di raggiungere tale limite, se il blocco richiesto ha dimensioni elevate (>128KByte)
  - malloc() richiede la creazione di un blocco separato attraverso la system call mmap()

Allocazione in Windows

- Un processo può gestire più heap
  - Quando il processo viene creato, c'è uno heap di base, usato da malloc
  - Altri heap possono essere creati e distrutti per implementare politiche particolari

HeapCreate & HeapDestroy

- HANDLE HeapCreate(...)
  - Crea un nuovo heap, specificandone le opzioni e dimensioni iniziali e massime
  - Restituisce un identificatore opaco (handle) che lo rappresenta
- BOOL HeapDestroy(HANDLE h)
  - Rilascia tutto lo spazio di indirizzamento associato allo heap indicato
  - Rende inaccessibile tutte le eventuali aree di memoria precedentemente ottenute da questo heap

HeapAlloc & HeapFree

- void *HeapAlloc(HANDLE h, DWORD options, SIZE_T s)
  - Alloca, dalla regione dello heap, un'area di dimensione s
  - Restituisce il puntatore relativo o NULL
- BOOL HeapFree(HANDLE h, DWORD options, void* ptr)
  - Rilascia un blocco precedentemente allocato
  - Restituisce FALSE in caso di errore