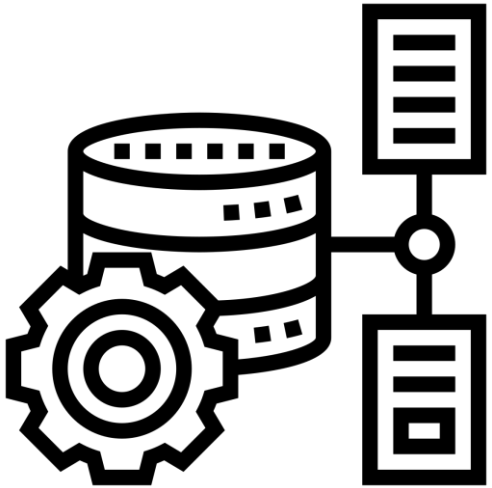


Allocazione della memoria

2024-25

Argomenti

- Architettura di elaborazione
- Spazio di indirizzamento
- Modello di esecuzione di un programma
- Stack e heap
- Puntatori e loro utilizzo

Programmi e loro esecuzione

- Un programma eseguibile è costituito da un insieme di istruzioni macchina, dati, valori di configurazione e controllo rappresentati come sequenze di numeri binari
  - Il significato delle istruzioni macchina è cablato all'interno del processore specifico per cui il programma è stato creato
  - Il significato delle informazioni restanti è contenuto, in parte, nelle istruzioni macchina e, per la parte restante, dipende dal sistema operativo (se c'è) o da particolari caratteristiche del processore stesso

- Per poter essere eseguito, un programma deve essere caricato all'interno della memoria accessibile al processore
  - Nei sistemi più piccoli (es., Arduino) questo avviene grazie ad hardware specifico che memorizza in modo semipermanente quanto necessario nella memoria flash
  - Nei sistemi più grandi, un modulo del sistema operativo (***loader***) si occupa di trasferire dal disco alla memoria RAM il contenuto del file eseguibile

Programmi e loro esecuzione

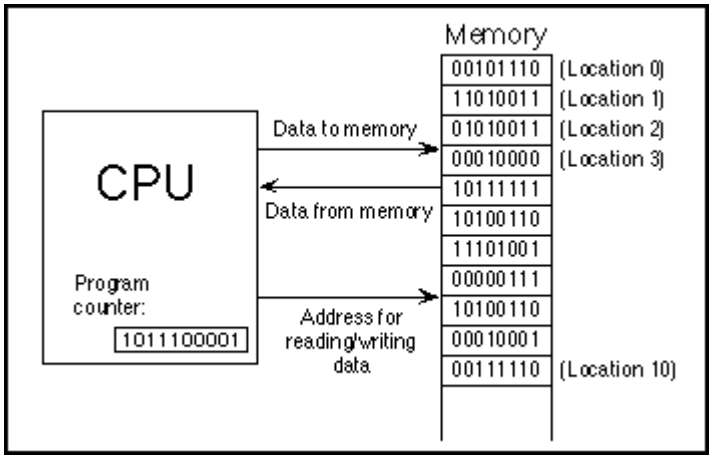
- Una volta che il programma è presente in memoria, può essere eseguito
  - Il processore preleva dalla memoria, una alla volta, le istruzioni macchina del programma e provvede ad eseguirle

- L'esecuzione di un'istruzione determina quali effetti collaterali debbano avvenire e quale debba essere la prossima istruzione da eseguire
  - Modifiche nelle informazioni contenute nel processore stesso e/o nella memoria
  - Eventuali interazioni con altre periferiche collegate al processore stesso (I/O)

- Il modello base di esecuzione prevede l'alternarsi continuo di tre fasi
  - **`Fetch`** - Un'istruzione viene trasferita dalla memoria all'interno del processore
  - **`Decode`** - L'istruzione prelevata viene trasformata in comandi da eseguire
  - **`Execute`** - I comandi vengono eseguiti

Programmi e loro esecuzione

- Il processore fa riferimento ad una specifica cella di memoria indicandone l'**indirizzo**
  - Uno speciale registro presente nella CPU (IP) "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, …)

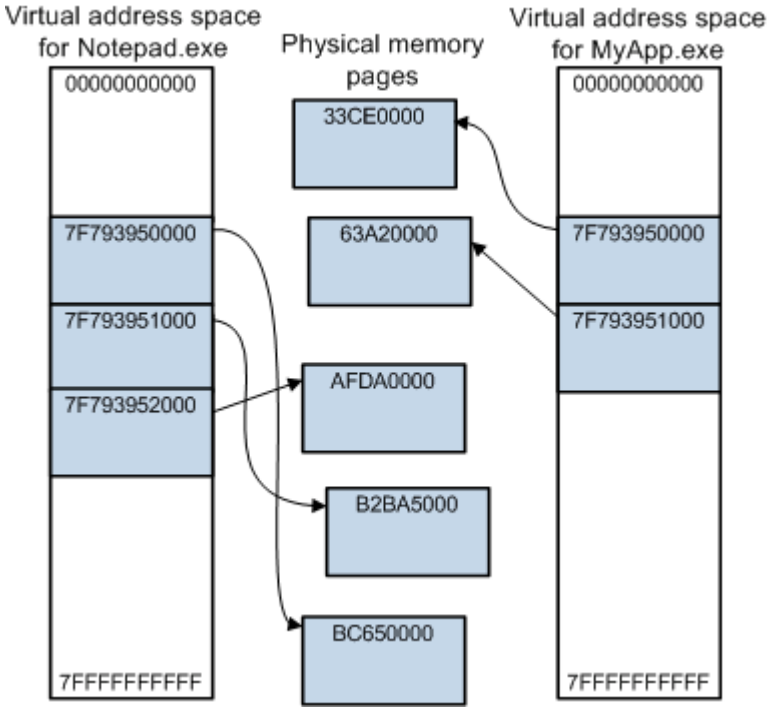


Spazio di indirizzamento

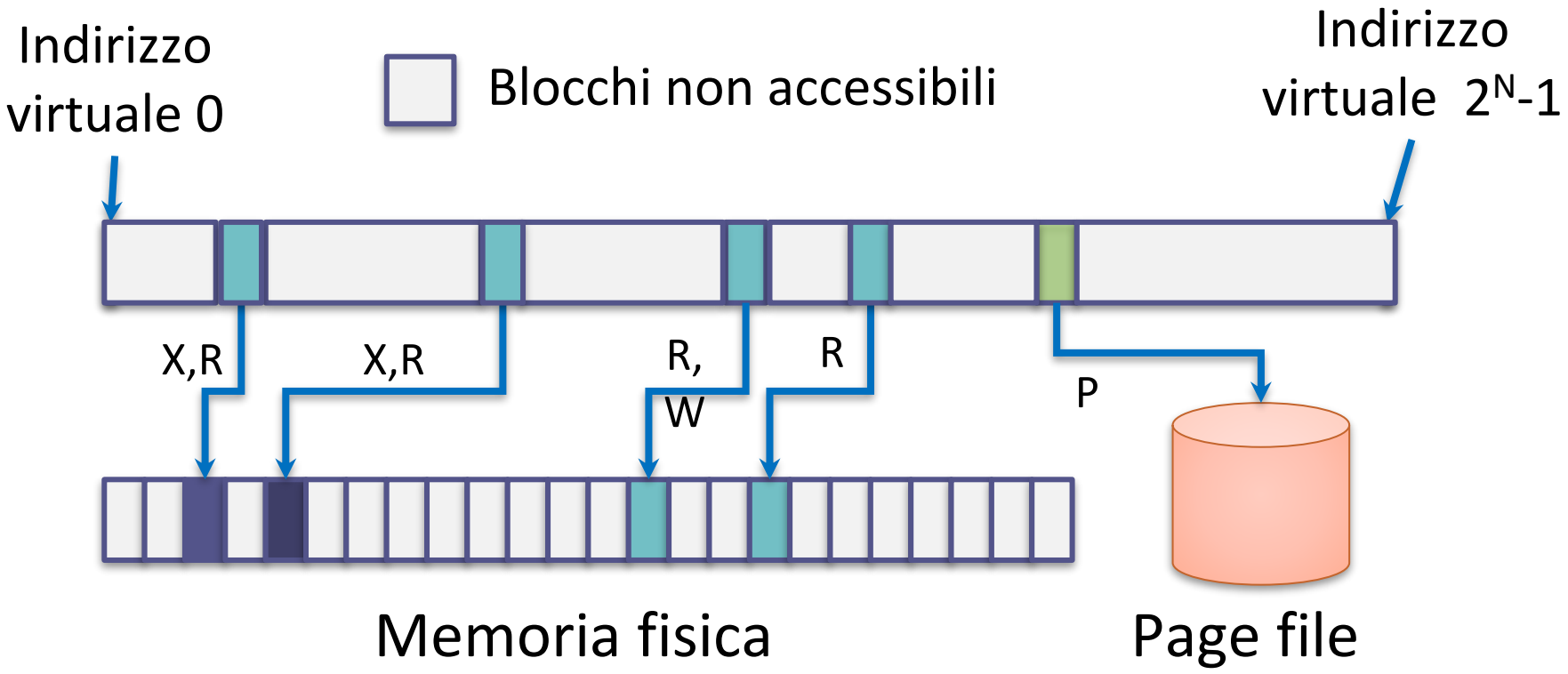
- L'esecuzione di un programma avviene nel suo **spazio di indirizzamento**
  - Sottoinsieme delle celle indirizzabili, gestito dal sistema operativo
- Dati, istruzioni, informazioni di controllo, … sono memorizzati come valori al suo interno
  - Lo spazio di indirizzamento può essere immaginato come un enorme array di byte, che possono essere letti individualmente, a coppie, a gruppi di 4, 8 alla volta, in base al parallelismo del processore (8, 16, 32 o 64 bit)
- Il numero di celle **potenzialmente** presenti nello spazio di indirizzamento dipende dal processore
  - Il numero di celle **effettivamente** presenti, è limitato dalla quantità di RAM disponibile ed è controllato dal sistema operativo
- Nel caso di CPU Intel x64, il processore opera a 64 bit
  - Ma lo spazio di indirizzamento si estende tra 0 e 2^{48}-1
  - All'interno di questo, solo una piccola parte di indirizzi è effettivamente presente

Spazio di indirizzamento

- Gli indirizzi che un programma utilizza non corrispondono - in generale - all'effettiva posizione che un dato assume in memoria
  - Il codice fa riferimento ad **indirizzi virtuali**
  - L'hardware utilizza **indirizzi fisici**
- Un apposito blocco hardware presente nel processore (**MMU - Memory Management Unit**) si occupa di tradurre l'indirizzo virtuale in indirizzo fisico
  - Tale traduzione garantisce che programmi diversi contemporaneamente in esecuzione non possano **interferire** tra loro
  - Consente di controllare quali **operazioni** siano **possibili** (lettura, scrittura, esecuzione)
  - Permette inoltre di scrivere programmi che manipolano **dati più grossi della memoria fisica** effettivamente presente



Spazio di indirizzamento

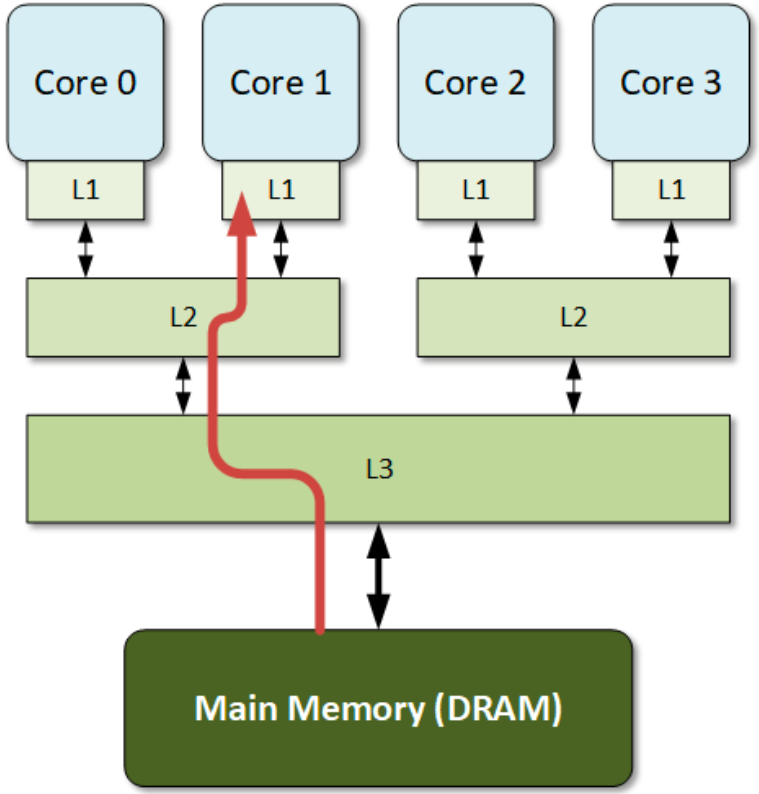


Gerarchia di memoria

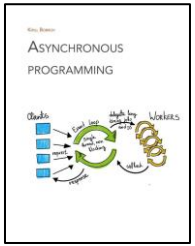
- Nei sistemi non elementari, il processore **non legge/scrive** direttamente dalla memoria RAM
  - Tra i due elementi viene interposta la memoria **cache**
  - Questa ha il compito di velocizzare l'accesso alle informazioni cercando di anticipare il bisogno di informazioni da parte della CPU
- Si basa sul **principio di località** dei programmi: se è stato appena fatto accesso all'indirizzo `I`, è molto probabile che venga fatto subito dopo accesso all'indirizzo `I+Δ` (con Δ molto piccolo)
  - La memoria cache si basa su dispositivi il cui tempo di accesso è molto più rapido di quello offerto dalla memoria RAM tradizionale, ma dotati di una capacità molto inferiore (pochi KB/MB)

Gerarchia di memoria

- Spesso la memoria cache è organizzata su vari livelli, via via più capaci ma anche più lenti
  - L'hardware di sistema si occupa di tutti i trasferimenti, nascondendoli all'attenzione del programmatore
  - In alcune situazioni (programmazione concorrente) è fondamentale comprendere cosa realmente avvenga nel sistema e tenerne conto



Impatto sui tempi di esecuzione



System event	Actual Latency	Scaled Latency
One CPU cycle	0.4 ns	1 s
Level 1 cache access	0.9 ns	2 s
Level 2 cache access	2.8 ns	7 s
Level 3 cache access	28 ns	1 min
Min memory access (DDR DIMM)	~100 ns	4 min
Intel Optane DC persistent memory access	~350 ns	15 min
Intel Optane DC SSD I/O	< 10 µs	7 hrs
NVMe SSD I/O	~25 µs	17 hrs
SSD I/O	50-150 µs	1.5-4 days
Rotational disk I/O	1-10 ms	1-9 months
Internet SF to NYC	65 ms	5 years

da "*Asynchronous programming*" di Kirill Bobrov, 2020

Esecuzione e programmi di alto livello

- Utilizzando linguaggi di programmazione ad alto livello, i dettagli legati al meccanismo di esecuzione sono normalmente **invisibili**
  - Tuttavia, per poter garantire il livello di astrazione necessario, essi devono introdurre delle sovrastrutture che impattano sul codice che viene scritto dal programmatore
  - …e introducono un insieme di **vincoli** e **restrizioni** di cui è necessario essere pienamente coscienti

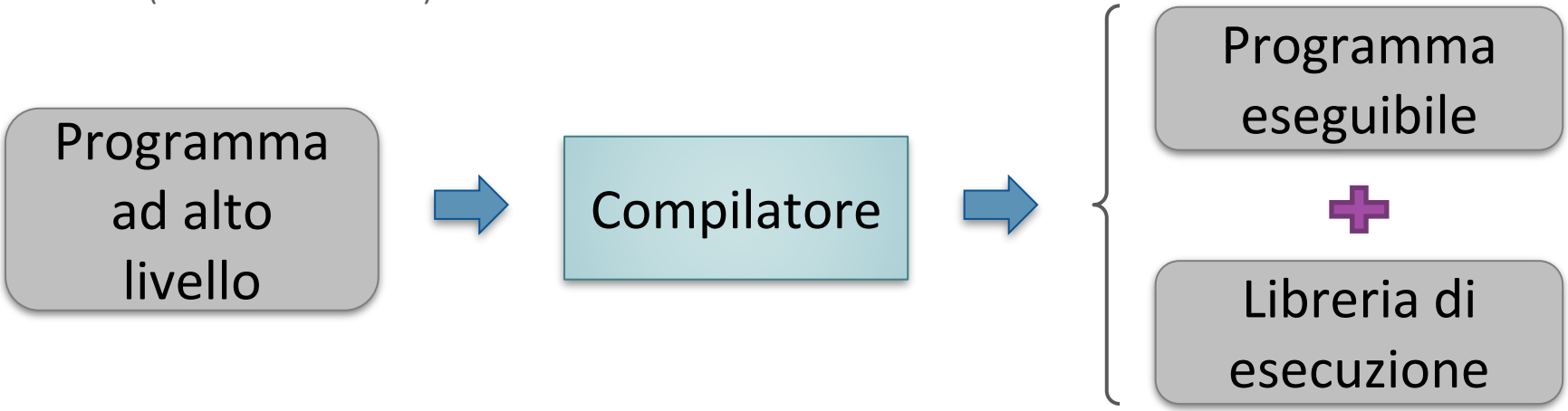
Modello di esecuzione di un programma

- Ogni linguaggio di programmazione propone uno specifico modello di esecuzione
  - Insieme di comportamenti attuati dall'elaboratore a fronte dei costrutti di alto livello del linguaggio

- Tale modello per lo più non corrisponde a quello di un dispositivo reale
  - E' compito del **compilatore** introdurre uno strato di adattamento che implementi il modello nei termini offerti dal dispositivo soggiacente
  - Questo strato è composto, da un lato, da una specifica modalità di traduzione dei costrutti di alto livello in istruzioni macchina e, dall'altro, da un insieme di funzionalità di supporto offerte sotto forma di libreria di supporto all'esecuzione (**run-time library**)

Livelli di astrazione

- La compilazione trasforma il programma sorgente in un nuovo programma dotato di un modello di esecuzione ed un insieme di istruzioni più semplice
  - Che può essere eseguito da una macchina fisica (CPU) o subire un'ulteriore trasformazione (esecuzione virtuale)



Librerie di esecuzione

- Offrono agli applicativi meccanismi di base per il loro funzionamento
  - Supportano le astrazioni del linguaggio di programmazione
  - Forniscono un'interfaccia uniforme tra i diversi S.O. per le funzioni ad essi demandate

- Costituite da due tipi di funzioni
  - Alcune, invisibili al programmatore, sono inserite in fase di compilazione per supportare l'esecuzione (es.: controllo dello stack, copia di aree di memoria, …)
  - Altre offrono al programmatore funzionalità standard, gestendo opportune strutture dati ausiliarie e/o richiedendo al S.O. quelle non altrimenti realizzabili (es.: malloc, fopen, …)

Modello di esecuzione nei linguaggi C e C++

- I programmi sono pensati come se fossero gli unici utilizzatori di un elaboratore, completamente dedicato loro (isolamento)
  - Le eventuali interazioni si riducono al più all'uso di risorse persistenti come il file system o le connessioni di rete
  - L'isolamento è reso possibile dal meccanismo di indirizzamento virtuale

- Un programma C/C++ assume di poter accedere a qualsiasi indirizzo di memoria presente nello spazio di indirizzamento
  - All'interno del quale può leggere o scrivere dati o dal quale può eseguire codice macchina
  - In realtà, gli indirizzi effettivamente accessibili sono molti meno, formano uno spazio sparso e non tutti consentono ogni tipo di operazione

Esecuzione sequenziale sincrona, senza limitazioni

- Un programma C/C++ è formato da un insieme di istruzioni eseguite, una per volta, nell'ordine indicato dal programmatore
  - Non ci sono limiti sul numero di istruzioni da eseguire, sul tempo richiesto alla loro esecuzione, né sulla memoria necessaria
- In realtà, tali limiti esistono e condizionano la scrittura dei programmi
  - Richiedendo al programmatore la comprensione del modello concettuale soggiacente
- All'interno del flusso principale di esecuzione è possibile attivare flussi di elaborazione secondari, detti ***thread***
  - Questi abilitano l'esecuzione concorrente (effettivamente parallela se la CPU è multicore) e introducono un ordine di grandezza nel livello di complessità della scrittura dei programmi

Flusso di esecuzione

- Il programma è costituito, come minimo, da un flusso di esecuzione il cui punto di partenza è predefinito
  - La funzione **`main()`** nel linguaggio C
  - I costruttori delle variabili globali in C++, seguiti dalla chiamata alla funzione **`main()`**

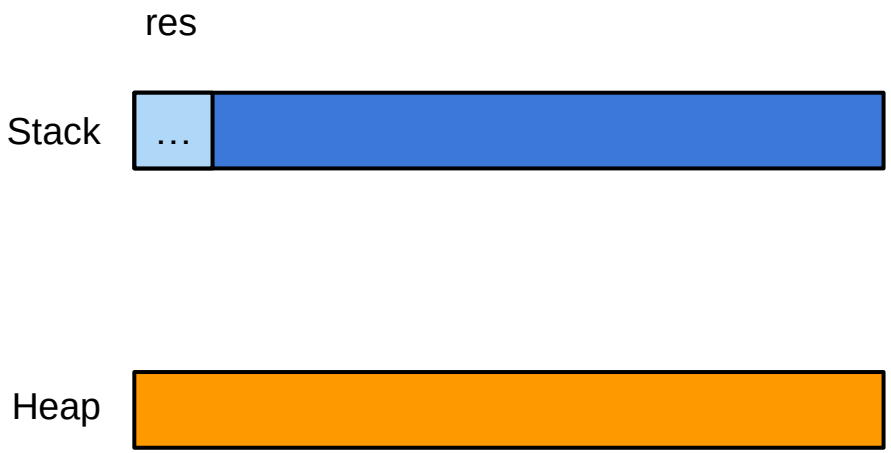
- Una struttura a pila, lo ***stack***, permette di gestire chiamate annidate tra funzioni
  - Essa supporta la ricorsione e l'uso di variabili locali
  - In C++, supporta anche la gestione strutturata delle eccezioni
- Una seconda struttura dati, lo ***heap***, offre supporto per gestire dinamicamente l'utilizzo della memoria
  - Secondo logiche non strettamente correlate al procedere dell'esecuzione

Stack

- Blocco di memoria allocato automaticamente all'avvio di un programma
  - Ha il compito di supportare l'esecuzione tenendo traccia della storia delle chiamate a funzione, degli argomenti passati ad ogni invocazione, dei valori di ritorno e delle variabili locali e temporanee presenti al loro interno
  - Viene utilizzato a partire da un estremo, espandendosi verso l'estremo opposto ogni volta che avviene una chiamata e contraendosi verso l'inizio quando una funzione ritorna
- Poiché ha dimensione finita, limita la profondità massima di ricorsione
  - Così come la dimensione totale delle variabili locali complessivamente utilizzabili
- Se, a seguito di un numero troppo elevato di chiamate o all'allocazione di variabili troppo grandi, si espande oltre il proprio limite, il sistema operativo (o il processore) interviene terminando l'esecuzione del programma
  - Stack overflow

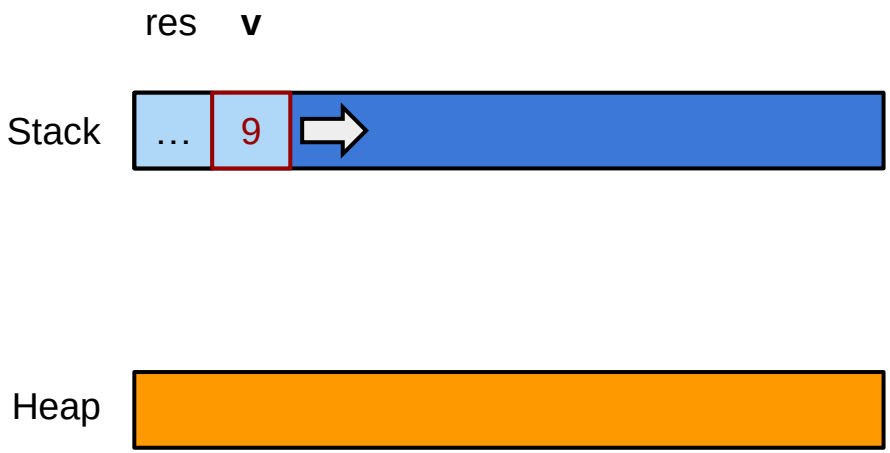
Stack

```
int f(int a) {
    int i = 5;
    if (a>0) {
        return a+i;
    } else {
        return a-1;
    }
}


int main() {
    int v = 9;
    v = f(v);
    return 0;
}
```



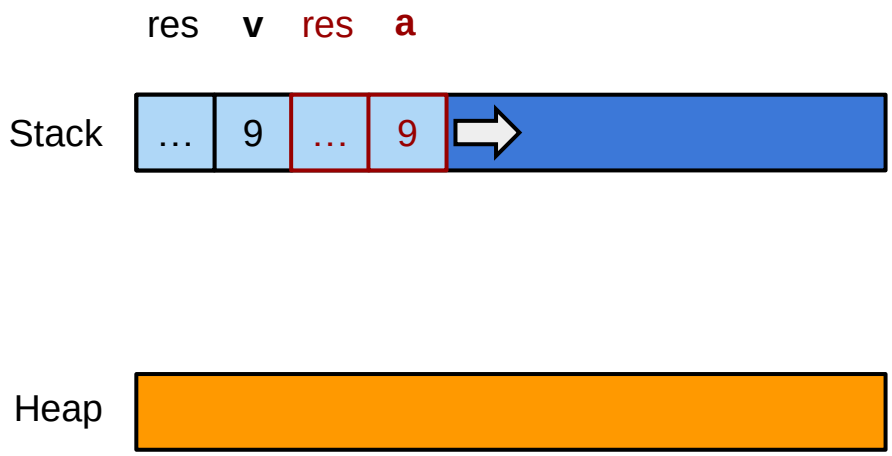
Stack

```
int f(int a) {
    int i = 5;
    if (a>0) {
        return a+i;
    } else {
        return a-1;
    }
}


int main() {
    int v = 9;
    v = f(v);
    return 0;
}
```



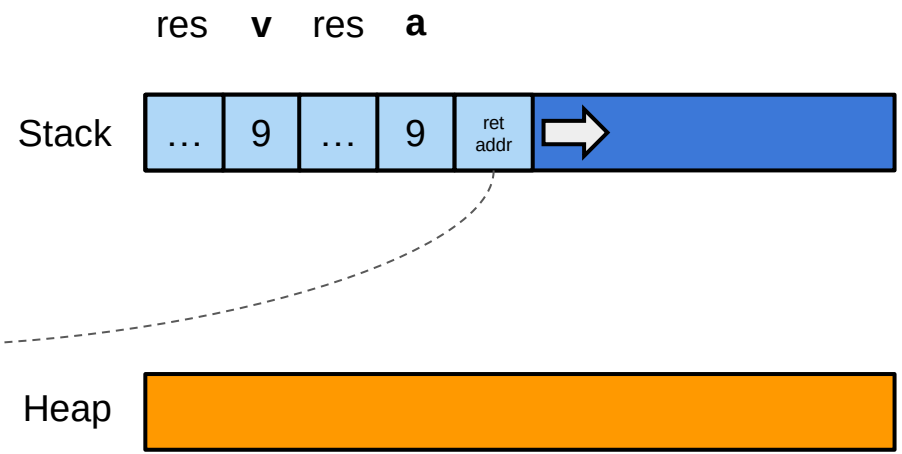
Stack

```
int f(int a) {
    int i = 5;
    if (a>0) {
        return a+i;
    } else {
        return a-1;
    }
}


int main() {
    int v = 9;
    v = f(v);
    return 0;
}
```



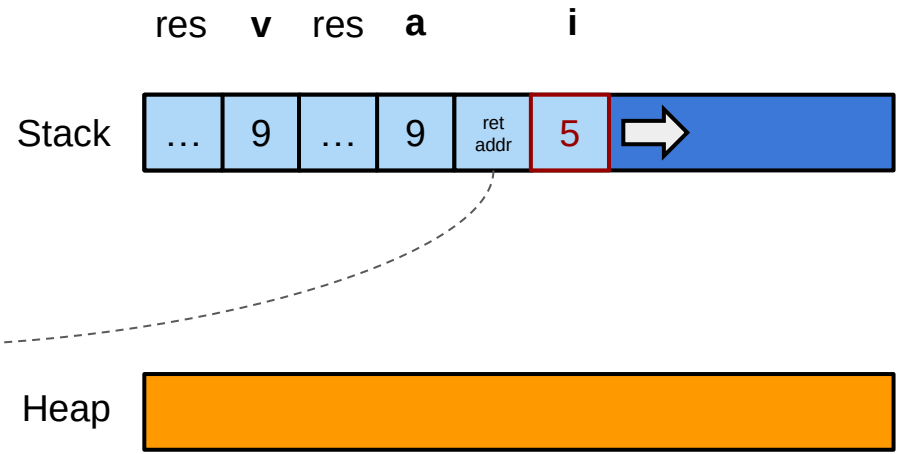
Stack

```
int f(int a) {
    int i = 5;
    if (a>0) {
        return a+i;
    } else {
        return a-1;
    }
}


int main() {
    int v = 9;
    v = f(v);
    return 0;
}
```



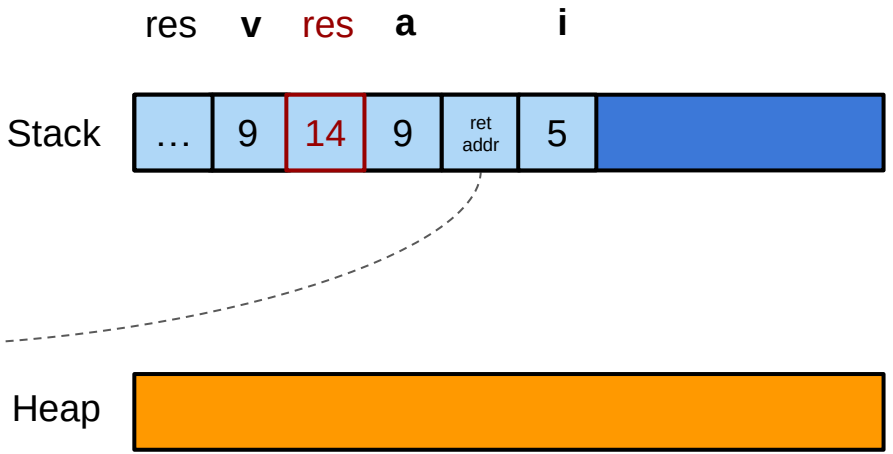
Stack

```
int f(int a) {
    int i = 5;
    if (a>0) {
        return a+i;
    } else {
        return a-1;
    }
}


int main() {
    int v = 9;
    v = f(v);
    return 0;
}
```



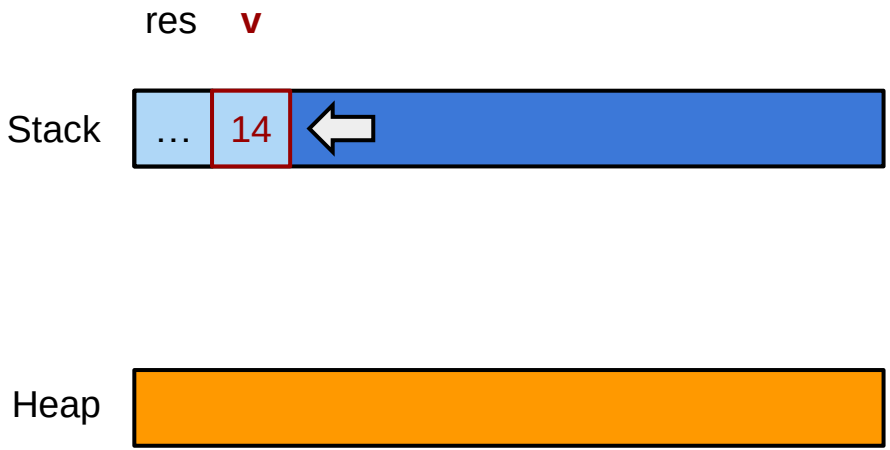
Stack

```
int f(int a) {
    int i = 5;
    if (a>0) {
        return a+i;
    } else {
        return a-1;
    }
}


int main() {
    int v = 9;
    v = f(v);
    return 0;
}
```



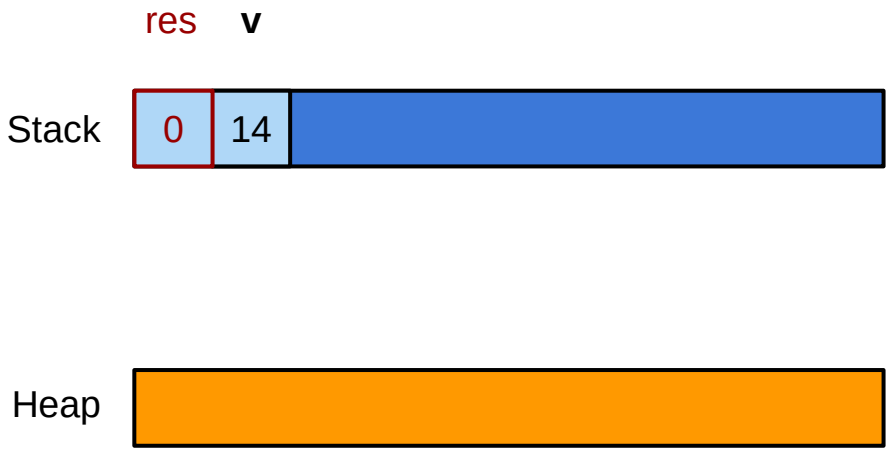
Stack

```
int f(int a) {
    int i = 5;
    if (a>0) {
        return a+i;
    } else {
        return a-1;
    }
}


int main() {
    int v = 9;
    v = f(v);
    return 0;
}
```



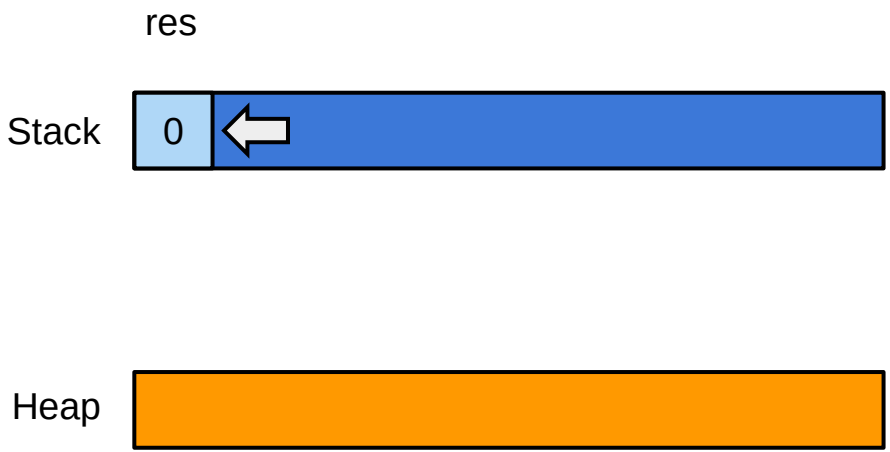
Stack

```
int f(int a) {
    int i = 5;
    if (a>0) {
        return a+i;
    } else {
        return a-1;
    }
}


int main() {
    int v = 9;
    v = f(v);
    return 0;
}
```



Stack

```
int f(int a) {
    int i = 5;
    if (a>0) {
        return a+i;
    } else {
        return a-1;
    }
}


int main() {
    int v = 9;
    v = f(v);
    return 0;
}
```



Stack

- L'allocazione ed il rilascio sullo stack sono estremamente efficienti, grazie alla particolare politica di espansione / contrazione adottata
  - Per contro, la durata dei valori memorizzati al suo interno è strettamente correlata alla durata dell'invocazione di un funzione
- Non è possibile memorizzare nello stack un dato che duri **più a lungo** della funzione in cui è stato allocato
  - Per questo motivo, il chiamante di una funzione ha la responsabilità di pre-allocare lo spazio in cui dovrà essere memorizzato il risultato prodotto dalla funzione chiamata
- Inoltre, poiché lo spazio complessivo dello stack è definito a priori, non è possibile memorizzare un dato la cui dimensione **superi** tale spazio
  - In generale, occorre limitare la dimensione massima del dato allocato ad una grandezza compatibile con il livello di profondità di chiamate e la dimensione degli altri dati che devono essere memorizzati complessivamente nello stack

Heap

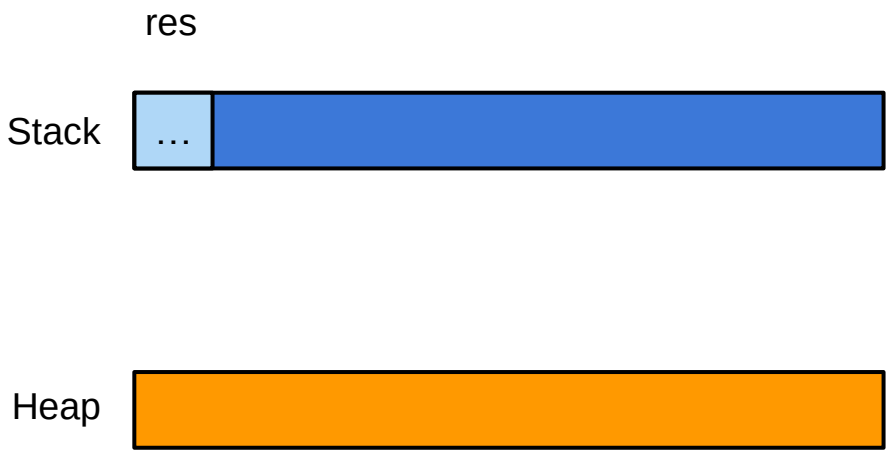
- Tutte le volte in cui un dato ha un ciclo di vita non collegato al tempo di esecuzione della funzione in cui è stato creato o la cui dimensione è cospicua oppure non nota in fase di compilazione, occorre richiederne la memorizzazione in una struttura a parte
  - I linguaggi C e C++ offrono un'apposita area, chiamata **Heap** o **Free Store**, all'interno della quale è possibile allocare e rilasciare blocchi di dimensione arbitraria (entro il limite della memoria disponibile)

- A differenza di quanto avviene per lo stack, le aree allocate sullo heap non sono associate, a livello di linguaggio sorgente, ad un nome di variabile
  - Si accede al loro contenuto esclusivamente tramite **puntatori**

- Inoltre, è responsabilità del programmatore rilasciare, prima o poi, la memoria allocata
  - Altrimenti si verificherà una perdita di memoria (memory leak) che, se reiterata, porta il sistema operativo a distruggere il processo

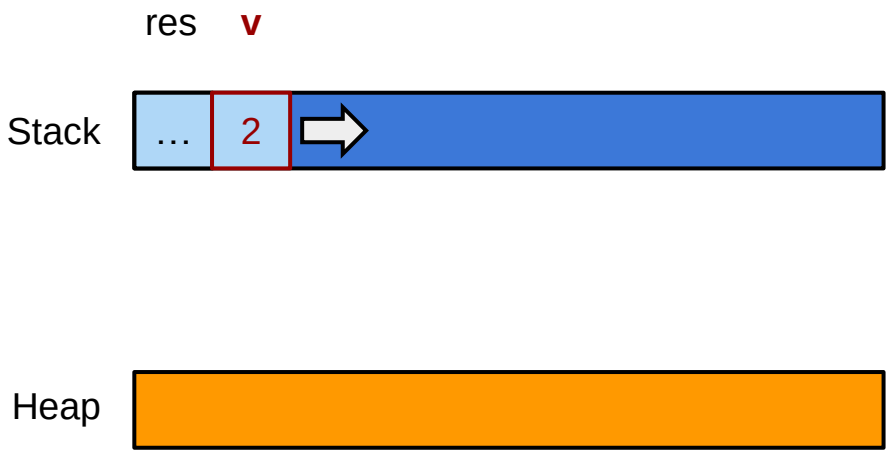
Heap

```
int* f(int a) {
    if (a>0) {
        return new int[a];
    } else {
        return nullptr;
    }
}

int main() {
    int v = 2;
    int* buf = f(v);
    //process buf
    delete[] buf;
    return 0;
}
```



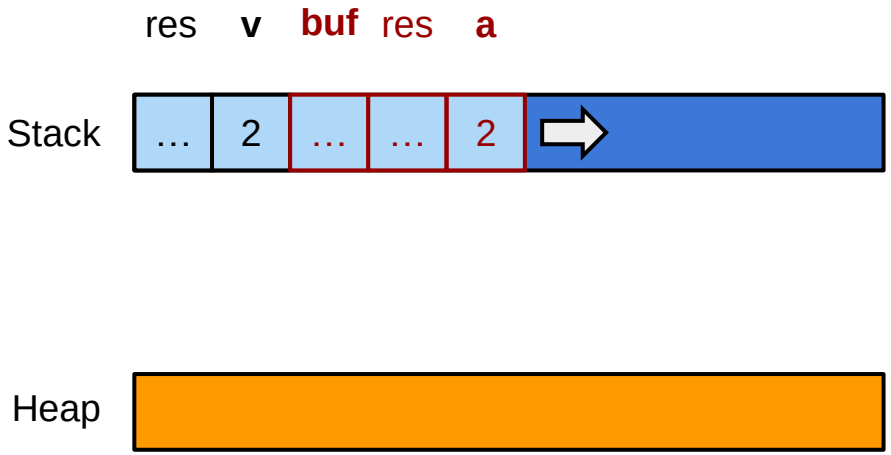
Heap

```
int* f(int a) {
    if (a>0) {
        return new int[a];
    } else {
        return nullptr;
    }
}

int main() {
    int v = 2;
    int* buf = f(v);
    //process buf
    delete[] buf;
    return 0;
}
```



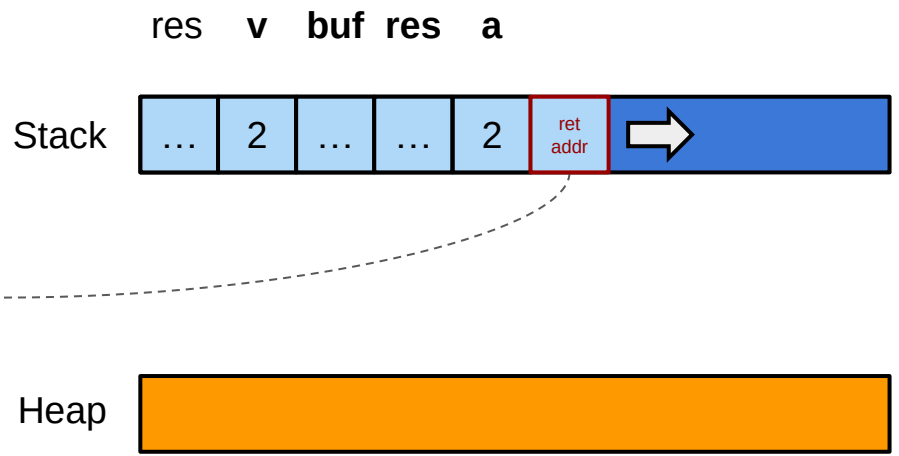
Heap

```
int* f(int a) {
    if (a>0) {
        return new int[a];
    } else {
        return nullptr;
    }
}

int main() {
    int v = 2;
    int* buf = f(v);
    //process buf
    delete[] buf;
    return 0;
}
```



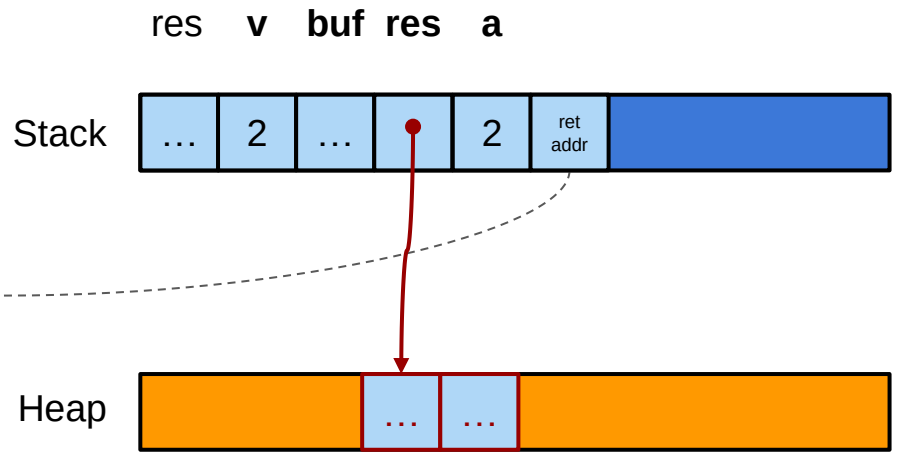
Heap

```
int* f(int a) {
    if (a>0) {
        return new int[a];
    } else {
        return nullptr;
    }
}

int main() {
    int v = 2;
    int* buf = f(v);
    //process buf
    delete[] buf;
    return 0;
}
```



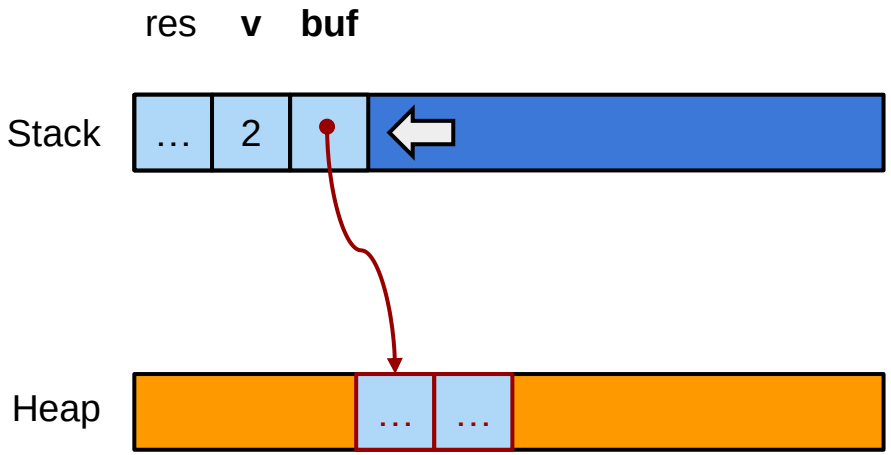
Heap

```
int* f(int a) {
    if (a>0) {
        return new int[a];
    } else {
        return nullptr;
    }
}

int main() {
    int v = 2;
    int* buf = f(v);
    //process buf
    delete[] buf;
    return 0;
}
```



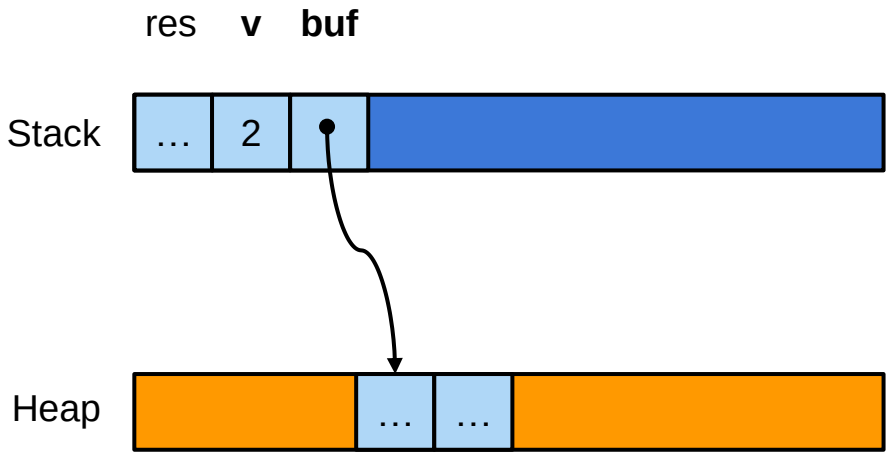
Heap

```
int* f(int a) {
    if (a>0) {
        return new int[a];
    } else {
        return nullptr;
    }
}

int main() {
    int v = 2;
    int* buf = f(v);
    //process buf
    delete[] buf;
    return 0;
}
```



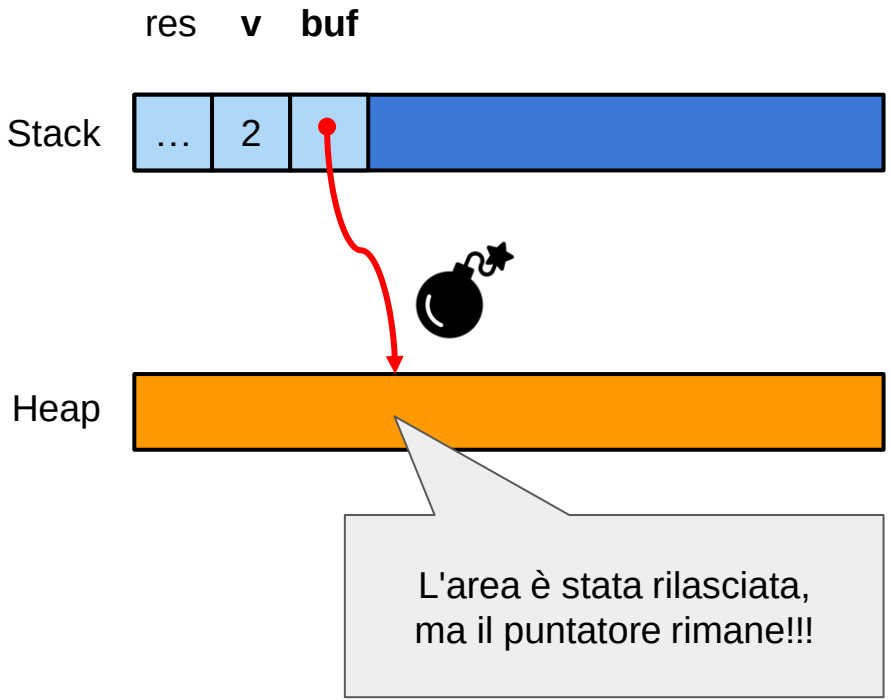
Heap

```
int* f(int a) {
    if (a>0) {
        return new int[a];
    } else {
        return nullptr;
    }
}

int main() {
    int v = 2;
    int* buf = f(v);
    //process buf
    delete[] buf;
    return 0;
}
```



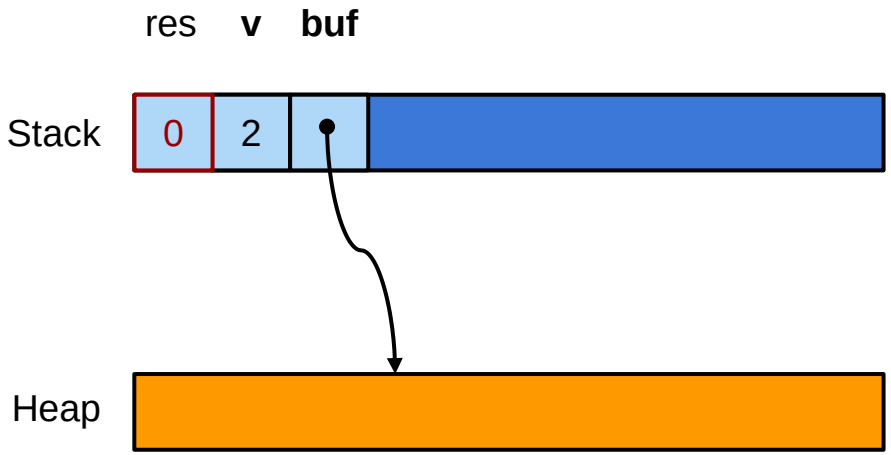
Heap

```
int* f(int a) {
    if (a>0) {
        return new int[a];
    } else {
        return nullptr;
    }
}

int main() {
    int v = 2;
    int* buf = f(v);
    //process buf
    delete[] buf;
    return 0;
}
```



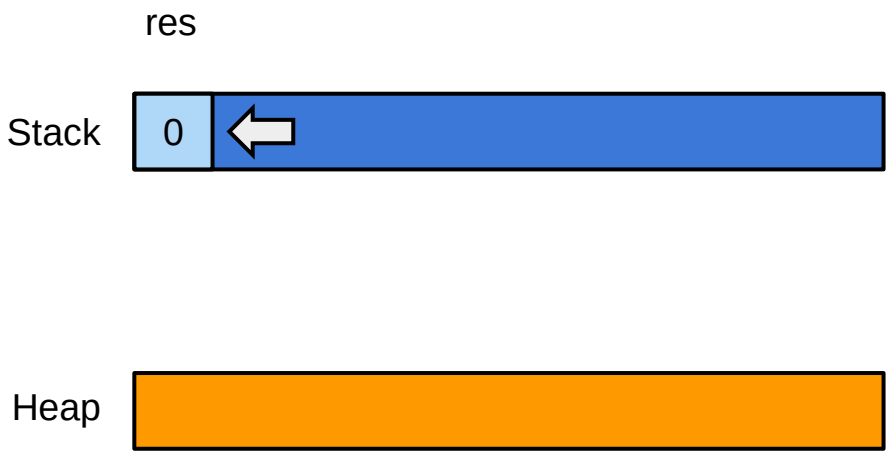
Heap

```
int* f(int a) {
    if (a>0) {
        return new int[a];
    } else {
        return nullptr;
    }
}

int main() {
    int v = 2;
    int* buf = f(v);
    //process buf
    delete[] buf;
    return 0;
}
```



Heap

```
int* f(int a) {
    if (a>0) {
        return new int[a];
    } else {
        return nullptr;
    }
}

int main() {
    int v = 2;
    int* buf = f(v);
    //process buf
    delete[] buf;
    return 0;
}
```



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
- 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

Organizzazione dello spazio di indirizzamento

- 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]
```

Ciclo di vita delle variabili

- Il modello di esecuzione dei linguaggi C/C++ distingue diverse **tipologie** di variabili
  - Globali, locali, dinamiche
- Ciascuna classe ha un proprio **ciclo di vita**
  - Intervallo di tempo in cui è garantito l'accesso alle informazioni contenute al loro interno

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

- Le 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 di 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 l'indirizzo di un blocco allocato dinamicamente
  - `void *malloc(size_t s)`
  - `void *calloc(int n, size_t s)`
  - `void *realloc(void* p, size_t s)`

- In caso di fallimento, viene restituito NULL
  - Se **`realloc(…)`** fallisce, il blocco originale non viene toccato e il puntatore **`p`** resta valido

Allocazione della memoria

- 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

Allocazione della memoria

- 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*

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 …[num_elements] / delete[ ]`
- Se il blocco non viene rilasciato si crea una perdita di memoria
  - Che, a lungo andare, provoca l'esaurimento dello spazio di indirizzamento
- Se il blocco viene rilasciato più volte o viene rilasciato con la funzione sbagliata
  - Si corrompono le strutture dati degli allocatori, con conseguenze imprevedibili

Puntatori

- La disponibilità di uno spazio di allocazione dinamico abilita l'implementazione di una vasta gamma di algoritmi e strutture dati che altrimenti risulterebbero di difficile implementazione
  - Tuttavia implica che il programmatore gestisca esplicitamente l'allocazione ed il rilascio dei blocchi al loro interno e ne manipoli il contenuto attraverso l'uso dei puntatori
- I puntatori sono uno strumento al tempo stesso estremamente potente ed estremamente pericoloso
  - Il loro utilizzo espone infatti il programmatore ad una vasta gamma di errori, le cui conseguenze portano - nella maggior parte dei casi - a **comportamenti non definiti** (undefined behavior), le cui conseguenze sono disastrose

Puntatori in C / C++

- Permettono l'accesso diretto ad un blocco di memoria
  - Tale blocco può appartenere ad altri oggetti
    - ```
      int A=10;
      int* pA = &A;
      ```
  - Essere allocato allo scopo
    - ```
      int* pB = new int{24};
      ```

- Possono essere esplicitamente invalidi
  - Il valore `0`
  - La macro `NULL ((void *)0)`
  - La parola-chiave `nullptr` (C++11 e superiori)

- Quando si usano i puntatori, occorre stabilire quali responsabilità/permessi sono associati al loro uso

Le ambiguità dei puntatori in C / C++

- Contiene un indirizzo valido?
- 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 altri conoscono lo stesso indirizzo?
- Viene usato come modo per esprimere l'opzionalità del dato?

Usi dei puntatori in C / C++

- Come strumento per accedere qui ed ora ad un'informazione contenuta in una altra struttura dati, senza doverla ricopiare
  - La responsabilità per la gestione della memoria del dato è totalmente esterna all'osservatore
  - Caso più semplice e alquanto frequente
  - Se l'accesso è in sola lettura, si antepone alla definizione del tipo puntato la parola chiave **`const`**

- Come modo per indicare ad una funzione dove depositare parte dei dati che essa deve calcolare
  - Anche in questo caso, la responsabilità è esterna all'osservatore
  - In C++11 e superiori, possono essere usate le tuple oppure i riferimenti per restituire più valori senza dover ricorrere ai puntatori

Puntatori come meccanismo per restituire più risultati

```
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;
}
```

Puntatori come meccanismo per accedere a blocchi di dati

- 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
  - Il C++11 introduce il tipo std::array<T, N> per mantenere sintatticamente l'informazione sulla dimensione del blocco

Puntatori come meccanismo per accedere a blocchi di dati

```
{
  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...
}
```

Puntatori come meccanismo per accedere a dati dinamici

- 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
  - E' il caso base di tutte le strutture dati dinamiche
  - Chi è responsabile del suo rilascio e quando va fatto?
  - Il C++11 introduce il concetto di smart pointer per gestire questa situazione in modo automatico e pulito

- Come modo per esprimere l'opzionalità del risultato
  - Responsabile del rilascio diventa necessariamente il chiamante
  - Il C++11 introduce il tipo generico std::optional<T> per evitare l'uso dei puntatori in questa situazione

Puntatori come meccanismo per accedere a dati dinamici

```
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;
```

Puntatori come meccanismo per implementare strutture dati articolate

- Strutture che richiedono la manipolazione di dati variamente collegati tra loro
  - 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 automatizzare il ciclo di vita del blocco puntato
  - Né meccanismi per associare una specifica semantica al puntatore

- In C++ sono state introdotte nuove astrazioni (riferimenti, smart pointer) che sollevano il programmatore da alcune responsabilità, facilitando la creazione di programmi più robusti
  - Ma richiedono in ogni caso una comprensione approfondita del problema e l'uso attento e coerente dei puntatori

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, usando la funzione duale di quella usata per la sua allocazione

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
- Rilasciare la memoria più volte **corrompe le strutture** usate dallo heap
  - ***Double free***

Rischi

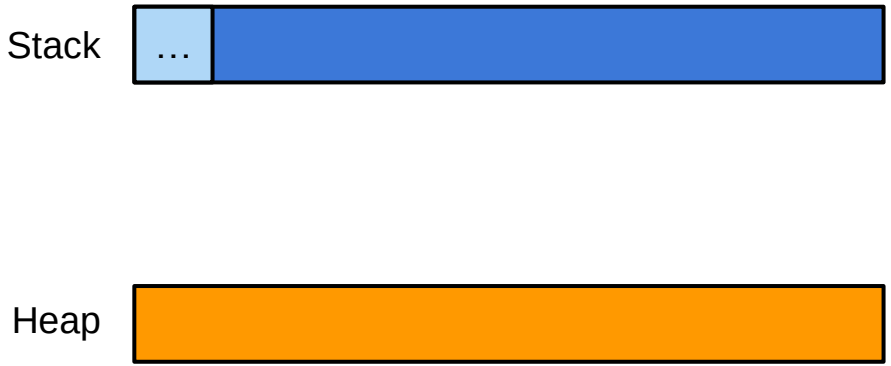
- Se si assegna ad un puntatore un indirizzo non mappato nello spazio di indirizzamento e si usa quel puntatore, viene generata una **interruzione del processore**
  - Il S.O. intercetta tale interruzione e termina il processo
- Se non si inizializza un puntatore e lo si usa, il suo contenuto potrebbe **puntare ovunque**
  - ***Wild pointer***
  - Può causare una violazione di accesso
  - Oppure corrompere un'area di memoria in uso ad altre parti del programma

Dangling Pointer

```
{
  char* ptr = nullptr;

  {  // 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
}
```

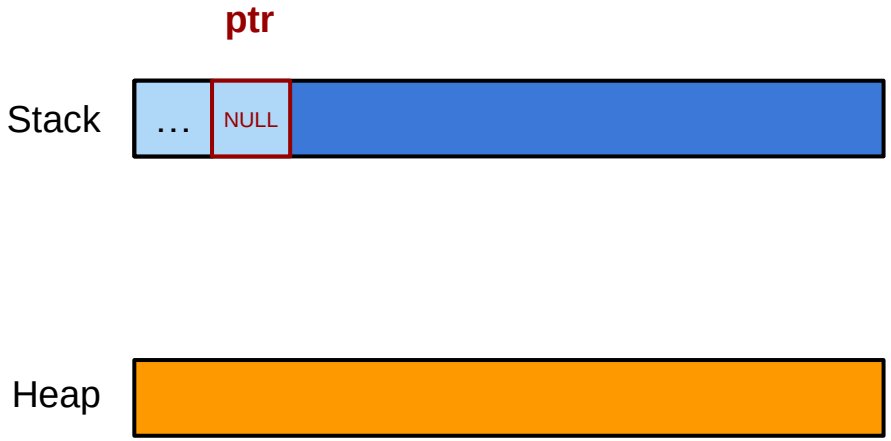
Dangling Pointer

```
{
  char* ptr = NULL;

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

  }

  printf("%c", *ptr);

}
```



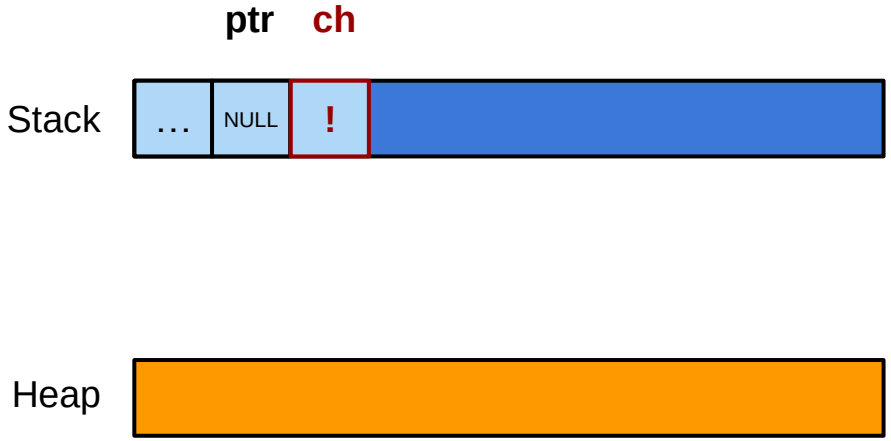
Dangling Pointer

```
{
  char* ptr = NULL;

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

  }

  printf("%c", *ptr);

}
```



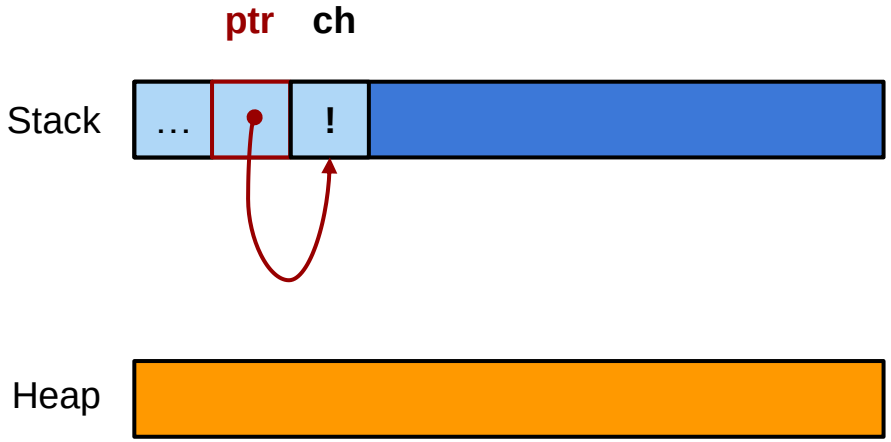
Dangling Pointer

```
{
  char* ptr = NULL;

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

  }

  printf("%c", *ptr);

}
```



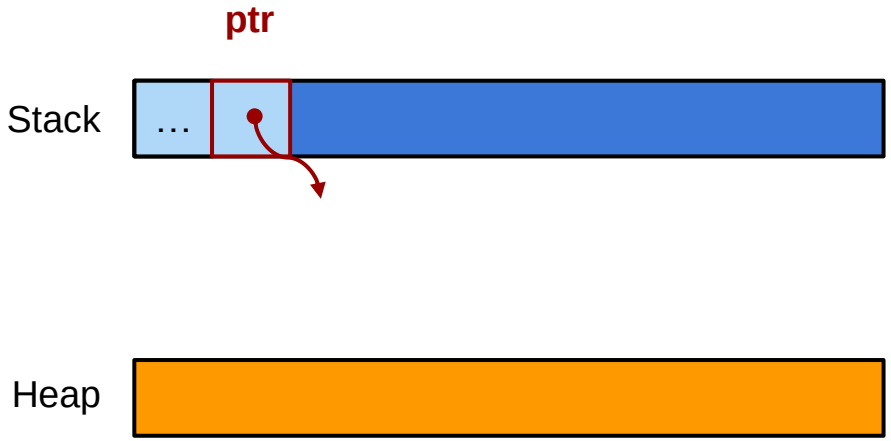
Dangling Pointer

```
{
  char* ptr = NULL;

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

  }

  printf("%c", *ptr);

}
```



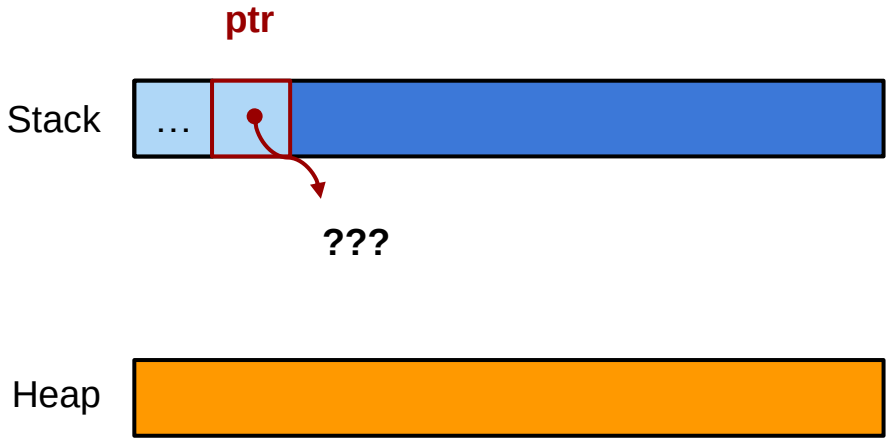
Dangling Pointer

```
{
  char* ptr = NULL;

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

  }

  printf("%c", *ptr);

}
```



Dangling Pointer

```
{
  char* ptr = NULL;

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

  }

  printf("%c", *ptr);

}
```

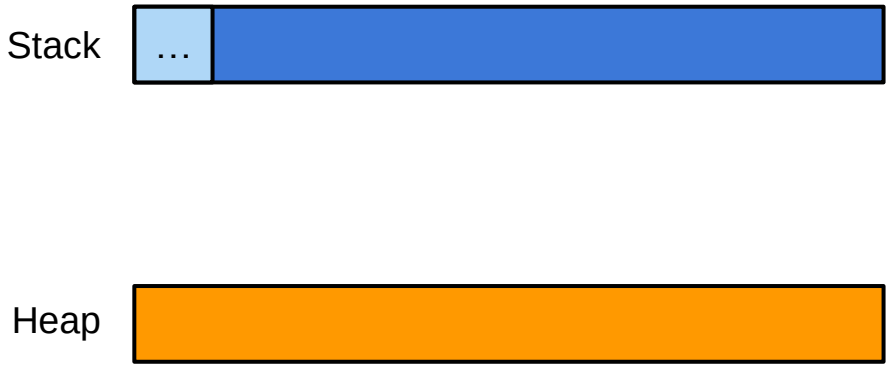


Memory leakage

```
{
  char* ptr = NULL;

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

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

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

}                                                  // Ne perdo le tracce
```

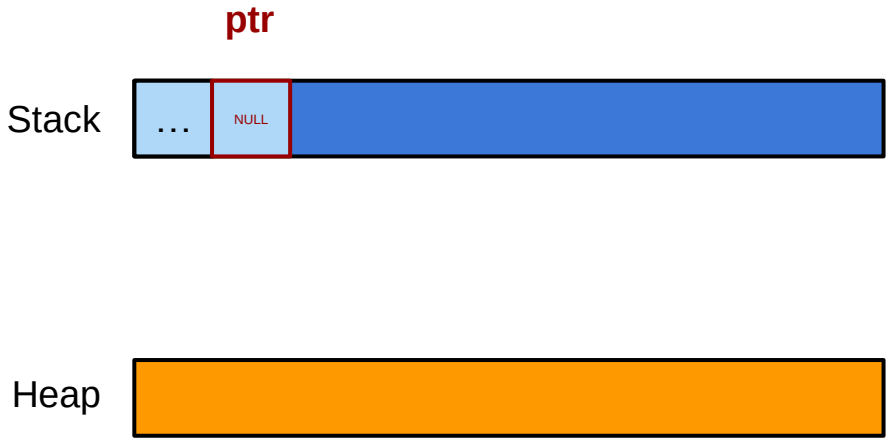
Memory leakage

```
{
   char* ptr = NULL;

   ptr = (char*) malloc(10);

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

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

}
```



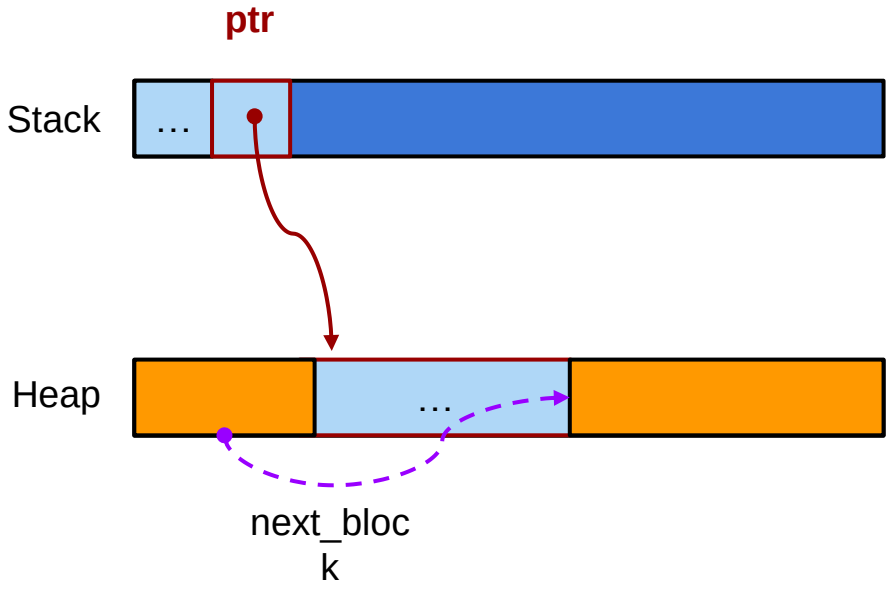
Memory leakage

```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);

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

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

}
```



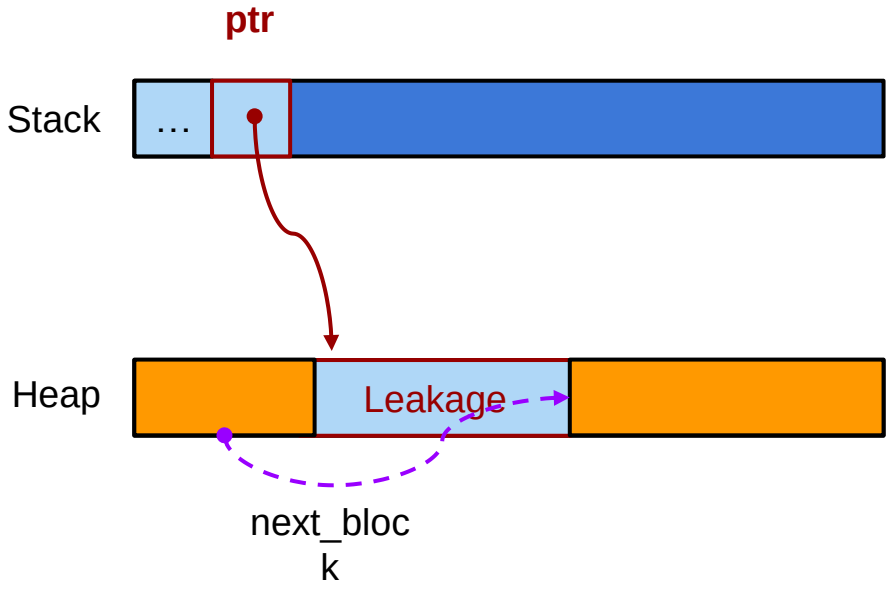
Memory leakage

```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);

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

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

}
```



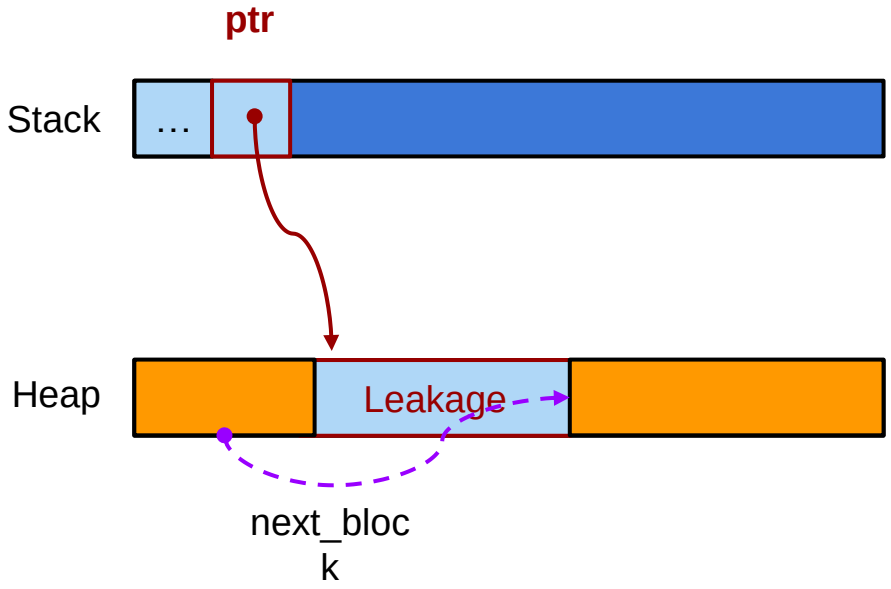
Memory leakage

```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);

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

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

}
```



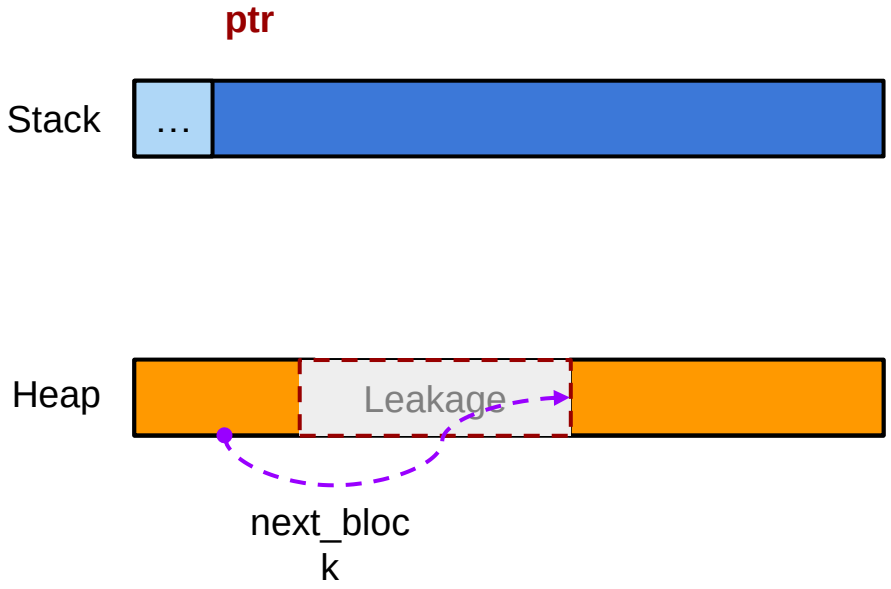
Memory leakage

```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);

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

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

}
```



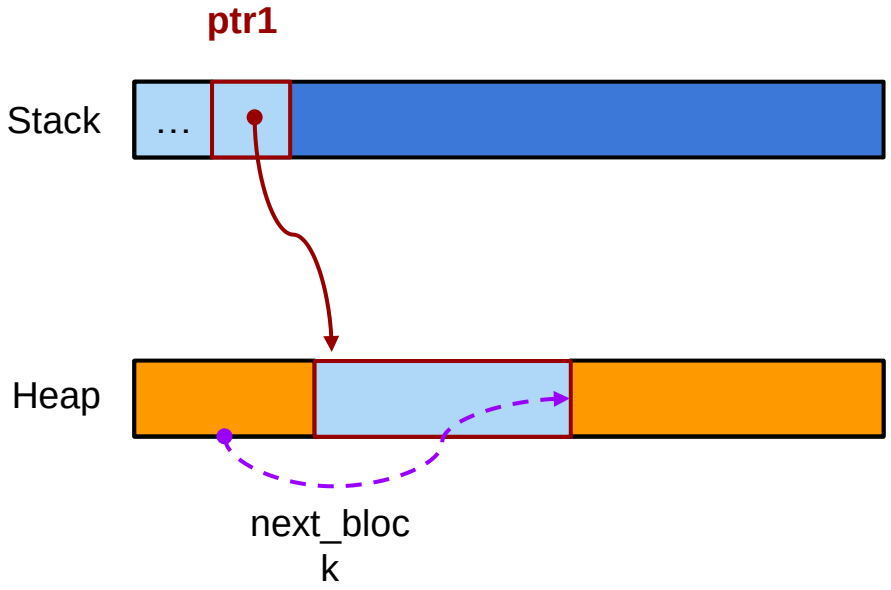
Memory leakage

```
{
  char* ptr = NULL;

  ptr = (char*) malloc(10);

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

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

}
```



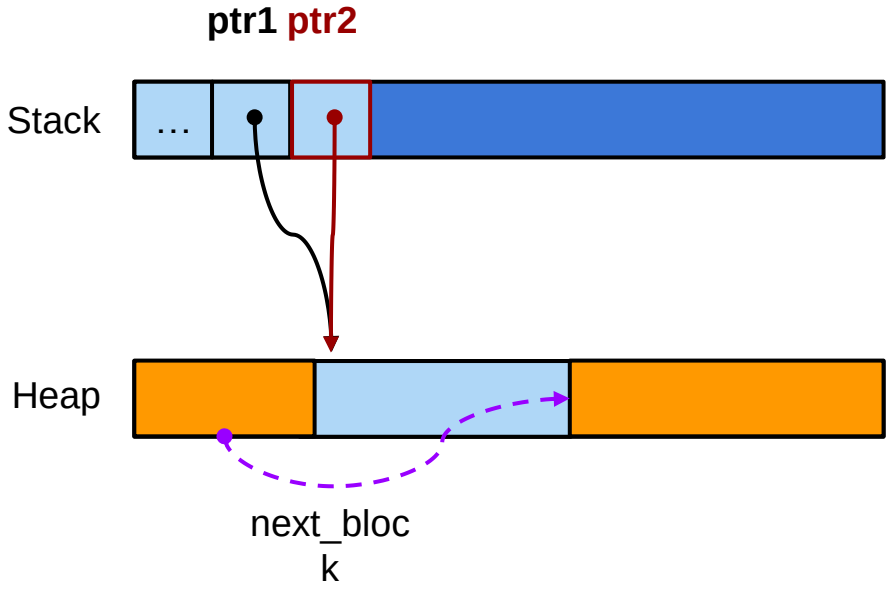
Double free

```
{
  char* ptr1 = malloc(10);

  char* ptr2 = ptr1;

  // use ptr1 and/or ptr2

  free(ptr1);

  free(ptr2);

}
```



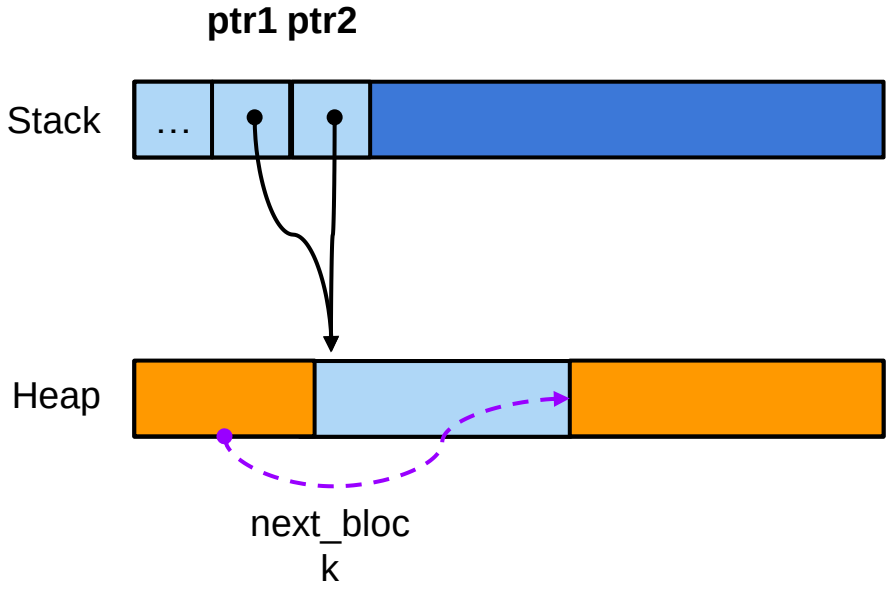
Double free

```
{
  char* ptr1 = malloc(10);

  char* ptr2 = ptr1;

  // use ptr1 and/or ptr2

  free(ptr1);

  free(ptr2);

}
```



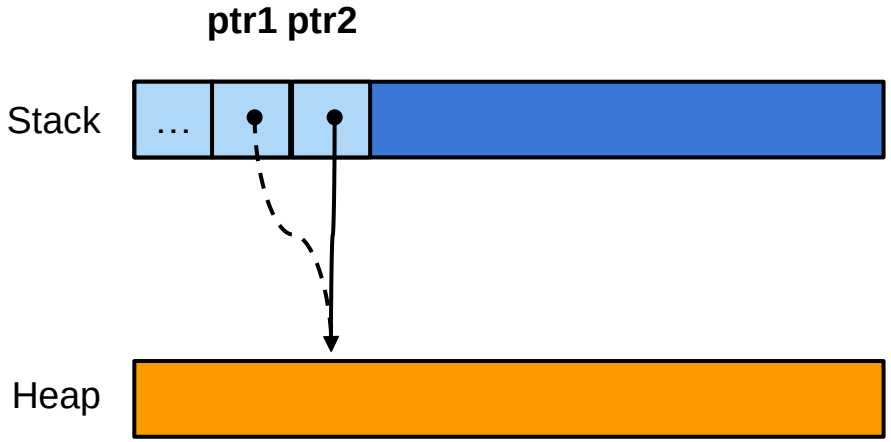
Double free

```
{
  char* ptr1 = malloc(10);

  char* ptr2 = ptr1;

  // use ptr1 and/or ptr2

  free(ptr1);

  free(ptr2);

}
```



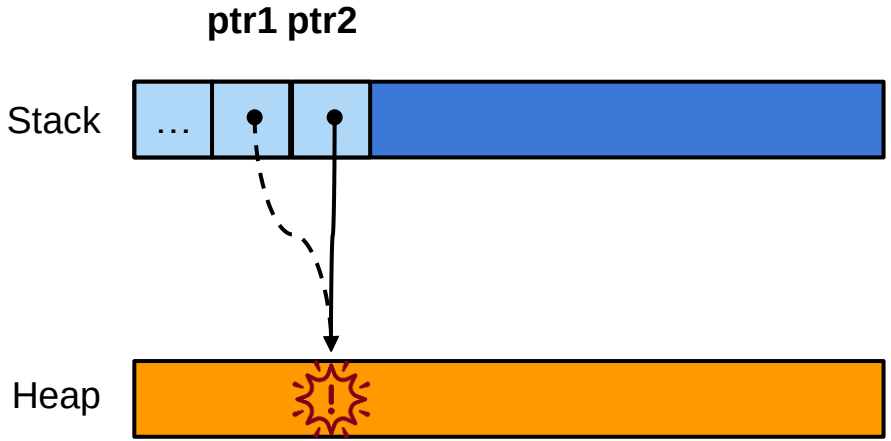
Double free

```
{
  char* ptr1 = malloc(10);

  char* ptr2 = ptr1;

  // use ptr1 and/or ptr2

  free(ptr1);

  free(ptr2);

}
```



Double free

```
{
  char* ptr1 = malloc(10);

  char* ptr2 = ptr1;

  // use ptr1 and/or ptr2

  free(ptr1);

  free(ptr2);

}
```



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 blocco

- Cosa capita se si effettua una **copia** di un puntatore?
  - Il secondo puntatore si trova coinvolto, suo malgrado, nel ciclo di vita del dato
  - Occorre introdurre un meccanismo per gestire efficacemente la semantica di un puntatore

Possesso della memoria

- Il vincolo di rilascio risulta problematico per via dell'**ambiguità** del concetto di puntatore all'interno del linguaggio
  - Dato un indirizzo non nullo, non è possibile distinguere se sia **valido o meno**, né **a quale area appartenga**, né se occorra o meno **rilasciarlo**

- 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**

Possesso 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?

Dipendenze

- Quando si introducono strutture dati complesse, è comune che tali strutture debbano appoggiarsi a blocchi di memoria ausiliari in cui memorizzare le informazioni che esse gestiscono
  - Un oggetto di tipo **`std::vector<T>`**, in C++, incapsula un array dinamico di dati di tipo generico **`T`**: tali dati possono essere aggiunti, rimossi, sostituiti, … tramite opportuni metodi sull'oggetto stesso
  - Tali blocchi ausiliari sono di proprietà dell'oggetto: vengono gestiti dai suoi metodi e devono essere rilasciati quando l'oggetto viene distrutto
- Analogamente, altre strutture dati possono mantenere riferimenti a oggetti del sistema operativo (handle a file, socket, timer, …)
  - Anche in questo caso le risorse corrispondenti risultano di proprietà dell'oggetto che le gestisce
- Collettivamente, questi tipi di dato prendono il nome di **dipendenze**

Dipendenze

- Il linguaggio C non offre nessun supporto sintattico per la gestione delle dipendenze e lascia al programmatore la completa responsabilità della loro gestione
  - Al contrario, il linguaggio C++ offre, per gli oggetti di tipo **`class`**, **`struct`** e **`union`** (!), la possibilità di specificare particolari funzioni membro dette rispettivamente **costruttori** e **distruttori**
- Un costruttore è una funzione membro che ha il compito di inizializzare un oggetto, provvedendo ad acquisire - se necessario - le sue dipendenze
  - Il distruttore è una funzione membro che viene invocata automaticamente dal compilatore quando l'oggetto raggiunge la fine della propria esistenza ed ha il compito di rilasciare correttamente le dipendenze possedute dall'oggetto stesso

Ma non basta scrivere un programma "giusto"?

- Sì, basterebbe se fosse possibile essere certi che lo è!
  - In realtà le cose sono molto più complesse
- Raramente i programmi sono scritti da una singola persona
  - Per lo più si appoggiano a librerie scritte da altri che hanno fatto opportune assunzioni su come debbano essere utilizzate, ma non è detto che tali assunzioni siano rispettate né (talvolta) conosciute
  - Spesso si lavora in gruppo e la comunicazione è imperfetta
- Al crescere della dimensione del programma, la quantità di particolari a cui occorre badare cresce molto più in fretta
  - Ed è estremamente probabile dimenticarsi delle proprie assunzioni
- Il 70% delle vulnerabilità rilevate all'interno di Windows sono dovute a problemi di gestione della memoria
  - https://en.wikipedia.org/wiki/Memory_safety

Perché questo problema non si verifica in altri linguaggi?

- Nella maggior parte dei linguaggi di alto livello il problema della gestione della memoria non si pone
  - Pur offrendo, a vario titolo, il concetto di puntatore
- Java, C#, Python, JavaScript, …, basano il proprio meccanismo di rilascio su un algoritmo di **Garbage Collection**
  - Che libera il programmatore dalla responsabilità di rilasciare esplicitamente la memoria dinamica allocata, in cambio della perdita di controllo su **quando** e **come** tale rilascio avvenga
- La maggior parte di tali linguaggi basa la propria strategia di gestione della memoria su una combinazione dei seguenti algoritmi
  - Reference counting
  - Mark and Sweep

Confronto

Gestione manuale in C/C++

- Il rilascio avviene invocando la funzione `free()` o l'operatore `delete`
- Il momento in cui avviene il rilascio è controllato dal programmatore
- In C++, gli oggetti dispongono di un distruttore che gestisce il rilascio esplicito delle dipendenze
- Il rilascio non comporta tempi supplementari di attesa, che sarebbero incompatibili con i sistemi in tempo reale
- Si possono verificare memory leak, doppi rilasci, dangling pointer, wild pointer,...

Gestione automatica (Java/C#/Python/Js)

- Il rilascio è eseguito automaticamente dal garbage collector
- Il momento in cui avviene il rilascio è controllato dal garbage collector, che viene eseguito periodicamente
- Il concetto di distruttore non è parte del linguaggio (ecc. Java, con il metodo `finalize()` ora deprecato)
- Quando si attiva la garbage collection, il programma si arresta fino al termine dell'algoritmo, mettendo a rischio il funzionamento di un sistema in tempo reale
- Non possono verificarsi errori sulla gestione della memoria

Tecniche di sopravvivenza

- Utilizzo di strumenti per la diagnosi dei processi
  - Valgrind Memcheck (Linux, MacOS, Android - https://www.valgrind.org/)
  - Dr.Memory (windows - http://drmemory.org/)

- Incapsulamento dei puntatori in apposite strutture dati che, sintatticamente, hanno lo stesso *pattern* di uso di un puntatore, ma esplicitano la semantica d'uso
  - smart pointer
  - Iterator
  - Span
  - Optional
  - Tuple
  - Costrutti di libreria (std::array, std::vector, …)

- Utilizzo di linguaggi "memory safe", come Rust

Per saperne di più

- Memory Management Algorithms and Implementation in C/C++
  - Bill Blunden, Wordware Publishing, Inc., 2003, ISBN 1-55622-347-1
  - Trattazione esaustiva, anche se datata, dei diversi algoritmi di allocazione della memoria, con i relativi limiti e prestazioni
- Understanding the Memory Layout of Linux Executables
  - https://gist.github.com/CMCDragonkai/10ab53654b2aa6ce55c11cfc5b2432a4
  - Approfondimento sulla struttura di un processo in Linux
- How to Manage Memory with Python
  - https://www.squash.io/how-to-manage-memory-with-python/
  - Descrizione dei meccanismi utilizzati dagli interpreti Python per la gestione della memoria