ALLOCAZIONE DELLA MEMORIA

Lo Stack e l'Heap.

Roberto Nardone

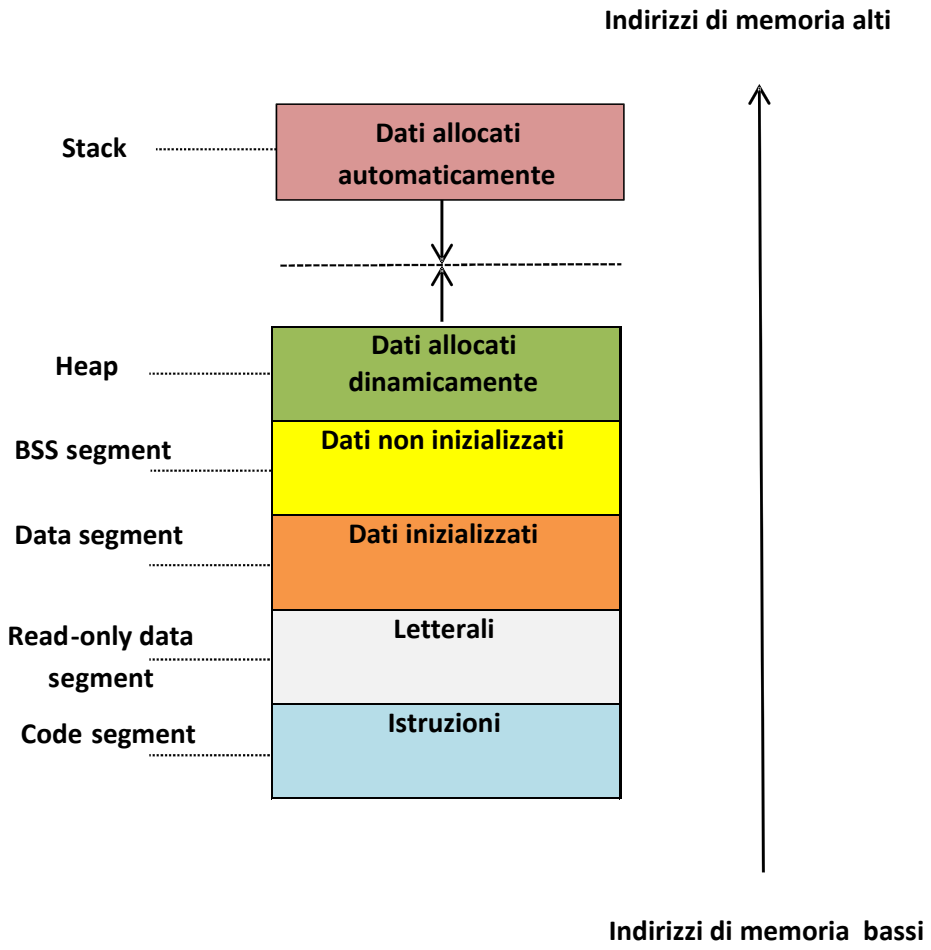
OUTLINE

- Programmi e memoria
  - Tipi di allocazioni della memoria
  - Stack
  - Heap

PROGRAMMI E MEMORIA

- Un programma e i dati su cui andrà ad operare vengono memorizzati nella memoria centrale del calcolatore.
- Il sistema operativo si occupa dell'assegnazione dell'area di memoria da destinare ai programmi e i loro dati.
- Tale memoria è suddivisa in **segmenti**, ed ogni segmento ha una funzione diversa.
- Partendo dagli indirizzi di memoria più bassi, i segmenti sono: *Code segment, Read-only data segment, Data segment, BSS segment, Heap e Stack.*

SEGMENTI DI PROGRAMMA IN MEMORIA[1]



SEGMENTI DI PROGRAMMA IN MEMORIA[2]

- **Code segment** e **Read-only data segment** sono aree di memoria la cui dimensione e contenuto è determinata dal compilatore e non cambiano durante l'esecuzione del programma.

- **Data** e **BSS segment** sono anch'esse aree di dimensione nota a tempo di compilazione, ma i dati al loro interno possono essere alterati.

- **Heap** e **Stack** riguardano il comportamento a tempo di esecuzione.

ALLOCAZIONE IN MEMORIA

- Il termine **allocazione** indica l'assegnazione di un blocco della memoria del calcolatore ad una variabile o più in generale ad un programma.

- L'allocazione della memoria può seguire tre diverse modalità:
  - allocazione automatica
  - allocazione statica
  - allocazione dinamica

ALLOCAZIONE AUTOMATICA

- Si ha allocazione automatica di un blocco di memoria quando in una funzione (**main()** compreso) viene definita una variabile locale.
- Il blocco viene creato a **tempo di compilazione** nello **stack**.
- La dimensione non sarà modificabile a **run-time**
- Il blocco verrà rilasciato **automaticamente** quando termina la funzione in cui è definita la variabile.

ALLOCAZIONE STATICA

- Si ha allocazione statica quando viene definita una variabile esterna a tutte le funzioni o dichiarata con la clausola **static**.
- Il blocco di memoria corrispondente viene creato a **tempo di compilazione** nel **Data segment**.
- La dimensione del blocco non è modificabile a **run-time**.
- Il blocco non è rilasciabile per tutta la durata del programma.

ALLOCAZIONE DINAMICA

- L'allocazione dinamica di un blocco di memoria avviene attraverso apposite funzioni su richiesta del programmatore.
- Il blocco di memoria viene allocato a **run-time** nell'**heap**.
- La dimensione è modificabile a **tempo di esecuzione** a seconda delle esigenze del programma.
- Il blocco allocato è rilasciabile **solo esplicitamente** dal programmatore attraverso una apposita funzione.

LO STACK

- Lo **stack** è una speciale regione della memoria del calcolatore che memorizza temporaneamente le variabili create da ciascuna funzione, **main()** compreso.

- Lo **stack** è un'area di memoria gestita **automaticamente** dalla CPU (c'è uno **stack pointer** memorizzato in un registro della CPU che punta all'indirizzo del primo elemento dello stack).

- L'accesso allo stack segue la politica **LIFO** (**L**ast **I**n **F**irst **O**ut) e ogni elemento dello stack è detto **stack frame.**

STACK FRAME

- Quando una funzione dichiara una variabile, viene allocato spazio nello stack per ospitarla e la variabile viene inserita (**push**) in testa allo stack.

- Ogni **stack frame** contiene tutte le variabili della funzione definite in un blocco compreso fra parentesi graffe.

- In fase di esecuzione, lo **stack frame** in testa è quello relativo al blocco correntemente in esecuzione, mentre i frame successivi corrispondono ai blocchi esterni.

- All'uscita dalla funzione, tutti gli stack frame ad essa riferiti vengono deallocati.

ESEMPIO[1]


```
#include<iostream>

using namespace std;

int main()
{
        int x=2;
        {
                int y=x+2;
                {
                        int z=y+2;
                }
        }
        return 0;
}
```

Stack: {x}

Stack: {y} {x}

Stack: {z} {y} {x}

Stack: {y} {x}

Stack: {x}

Stack: {}

ESEMPIO[2]

- Nell'esempio precedente le variabili **x**, **y**, **z** sono immagazzinate nello stack fin tanto che il loro ambito di visibilità è attivo.

- La variabile che sopravvivrà per più tempo nello stack è quella contenuta nel blocco più esterno (**x**).

- Al di fuori del proprio ambito di visibilità le variabili vengono distrutte e l'area di memoria corrispondente liberata.

- Da questo deriva il concetto di **variabili automatiche** (le variabili locali lo sono per definizione), infatti non è richiesta alcuna istruzione per cancellarle dalla memoria.

STACK - RIEPILOGO

- Ospita le variabili locali di una funzione.
- La gestione di allocazione e deallocazione è automatica.
- Politica di accesso alla memoria di tipo LIFO.
- Una variabile sopravvive nello stack fin tanto che il suo ambito di visibilità è attivo.
- La dimensione dello stack è limitata e dipende dal linguaggio e dall'architettura usata. Quando si eccede il limite, una condizione di errore nota come **stack overflow** provoca la terminazione del programma.

HEAP

Come si può tenere in memoria una grande quantità di dati o estendere la vita delle variabili anche all'esterno del loro ambito di visibilità?

Per queste esigenze ci viene incontro un'altra area della memoria nota come l'**heap**.

L'**heap** è un'area di memoria allocata e deallocata **dinamicamente** su richiesta del programmatore, di dimensione più ampia dello stack e **permanente** rispetto agli ambiti di visibilità di blocchi e funzioni.

HEAP: ALLOCAZIONE DELLA MEMORIA

- L'allocazione dinamica della memoria viene effettuata utilizzando la parola chiave **new**, viceversa la deallocazione richiede la parola chiave **delete**.

- Quando alloco memoria dinamicamente, la alloco a tempo di esecuzione (a **run-time**), e non a tempo di compilazione come avveniva per lo stack.

- Ciò implica che non avrò una variabile associata all'area di memoria allocata dinamicamente.

DEALLOCAZIONE DELLA MEMORIA

Come faccio ad accedere all'area di memoria allocata?

- L'operatore **new** non si limita ad allocare un'area di memoria nell'**heap**, ma restituisce l'indirizzo della locazione di memoria (il **puntatore**) dove inizia quest'area.

- L'operatore **delete** deve essere utilizzato solo per un puntatore a memoria che è stato allocato con l'operatore **new**.

- Per non avere problemi di **memory leak** (porzioni dell'heap occupate, ma non più necessarie per l'autonomia del programma) ad ogni puntatore a **new** deve corrispondere un **delete**.

ALLOCAZIONE DINAMICA[2]


```
#include<iostream>

using namespace std;

int main()
{
        int *p;
        p= new int;

        *p=100;
        cout<<"Il valore puntato da p è: "<<*p<<endl;

        delete p;

        cout<<"Il valore puntato da p dopo l'operazione di delete è: "<<*p<<endl;

        return 0;
}
```

1. Dichiaro un puntatore ad interi;
2. Alloco dinamicamente memoria per un intero;
3. Il puntatore si riferisce all'area appena allocata.

Dealloco l'area puntata da p.

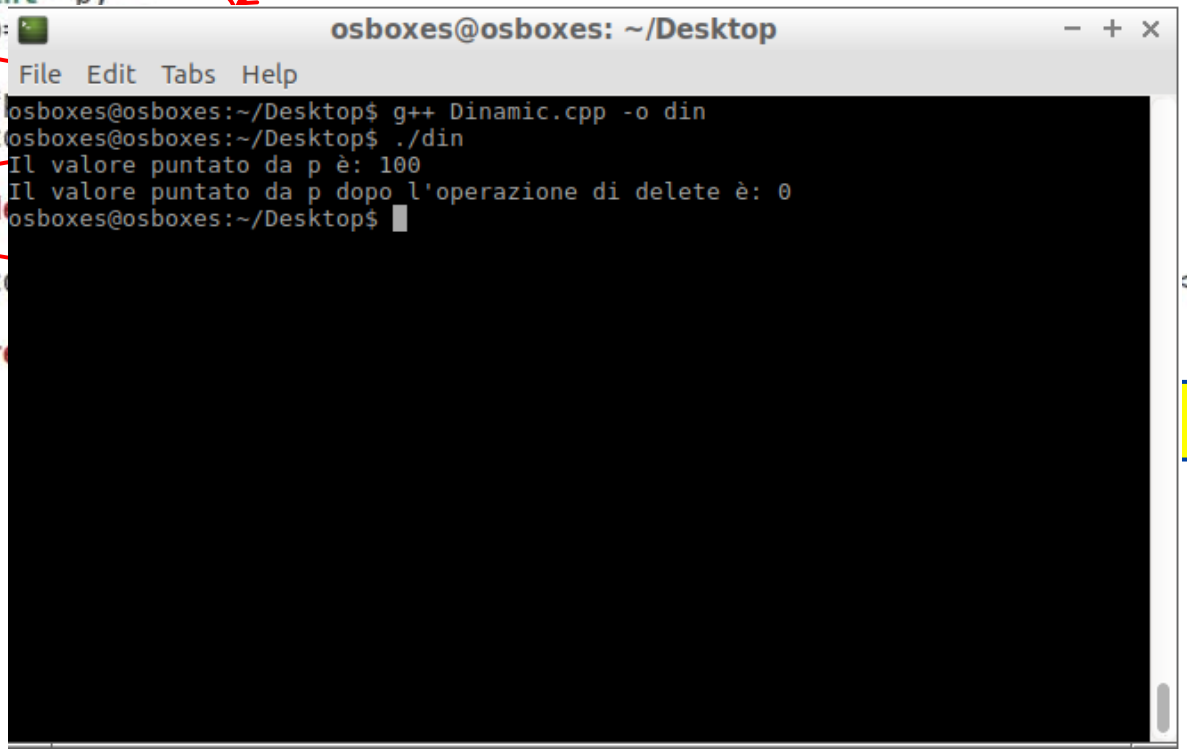
ALLOCAZIONE DINAMICA[2]


```
#include<iostream>

using namespace std;

int main()
{
        int *p;
        p=
        *
        C
        d
        c
        r
}
```

1. Dichiaro un puntatore ad interi;
2. Alloco dinamicamente memoria per un intero;
3. Il puntatore si riferisce all'area appena allocata.




```
<<endl;
```

ALLOCAZIONE DINAMICA[3]

- È possibile inizializzare direttamente la memoria allocata dinamicamente con l'operatore **new**.

- Nel caso di allocazione dinamica per una variabile intera, supponendo di voler inizializzare il valore a 100, si procede come segue:

```
int *p;
p=new int(100);
```

Se invece volessi allocare dinamicamente un array?

ALLOCAZIONE DINAMICA DI UN ARRAY[1]

- In un programma un array è tipicamente una struttura dati la cui dimensione può cambiare a tempo di esecuzione.

- Allocare un array a tempo di compilazione (**array statici**), comporta la definizione a priori di una dimensione per l'array.

- Tale dimensione durante l'esecuzione del programma potrebbe risultare insufficiente (non avrò spazio per allocare nuovi elementi) o eccessiva (occupazione inutile di memoria).

- Allocare gli array a **tempo di esecuzione** è possibile ed anche in questo caso ci viene in soccorso l'operatore **new**. Array allocati a **run-time** sono detti **array dinamici**.

ALLOCAZIONE DINAMICA DI UN ARRAY[2]

- Come per una variabile, anche per allocare un array dinamico andrà dichiarato un puntatore del tipo opportuno (quello degli elementi dell'array) che dovrà puntare all'area di memoria che contiene il primo elemento dell'array:

```
double *p;
int dim;
cin>>dim;
p=new double[dim];
```

- Nell'esempio **p** punta all'indirizzo del primo **double** memorizzato nell'array dinamico. La dimensione è scelta dall'utente a tempo di esecuzione.

ALLOCAZIONE DINAMICA DI UN ARRAY[3]

Non confondere il puntatore all'array con l'array stesso.

Il **puntatore** serve esclusivamente a conservare l'indirizzo di memoria dell'array per poterci accedere in un secondo momento.

Un array dinamico una volta allocato esiste fino alla fine del programma (o fino a quando non viene deallocato esplicitamente).

Puntatore e array dinamico hanno tempi di vita diversi e occupano aree di memoria diverse. Quindi potrebbe accadere che il puntatore non esiste più, mentre l'array è ancora in memoria.

Come si dealloca un array dinamico?

DEALLOCAZIONE DI UN ARRAY DINAMICO

- La deallocazione di un array dinamico avviene attraverso l'operatore **delete[]** specificando il puntatore all'array:

```
double *p;
int dim;
cin>>dim;
p=new double[dim];
delete[] p;
```

- L'operatore **delete[]** può essere applicato **solo** ad un array allocato con l'operatore **new**.

ESERCIZI

1. Realizzare un programma che dati 2 array di interi allocati dinamicamente, stampi a video il prodotto fra i due vettori.*

2. Realizzare un programma che dati 2 array di double allocati dinamicamente, memorizza in un terzo array gli elementi dei 2 array precedentemente inizializzati. **

* La dimensione degli array deve essere la stessa e impostata da tastiera dall'utente.

** v1=[1 2 3], v2[3 4 5] -> v3=[1 2 3 4 5]

ALLOCAZIONE DINAMICA IN C

- In C l'allocazione dinamica è gestita mediante le funzioni **`malloc, calloc, realloc`** e **`free`** di stdlib.h
- `void *calloc(size_t size)` : Alloca size byte nello heap. **La memoria viene inizializzata a 0**. La funzione restituisce il puntatore alla zona di memoria allocata in caso di successo e NULL in caso di fallimento
- `void *malloc(size_t size)` : Alloca size byte nello heap. La memoria non viene inizializzata. La funzione restituisce il puntatore alla zona di memoria allocata in caso di successo e NULL in caso di fallimento
- `void *realloc(void *ptr, size_t size)` : Cambia la dimensione del blocco allocato all'indirizzo ptr portandola a size. La funzione restituisce il puntatore alla zona di memoria allocata in caso di successo e NULL in caso di fallimento
- `void free(void *ptr)` : Dealloca lo spazio di memoria puntato da ptr.

ESEMPIO


```
#include <iostream>
#include <stdlib.h>
int main(){
        int num, i, *ptr, sum = 0;
        cout << "Numero di elementi: " ;
        cin >> num;
        ptr = (int*) malloc(num * sizeof(int));  //allocazione con malloc
        if(ptr == NULL) {
                cout << "Errore! Memoria non allocata.";
        }  else {
                cout << "Inserisci gli elementi dell'array: ";
                for(i = 0; i < num; ++i)    {
                        cin >> *(ptr + i);
                        sum += *(ptr + i);
                }
                cout << "Sum =" <<  sum << endl;
                free(ptr);
        }
}
```