

Corso “Programmazione 1”

Capitolo 08: Gestione Dinamica della Memoria

Docente: **Marco Roveri** - marco.roveri@unitn.it
Esercitori: **Giovanna Varni** - giovanna.varni@unitn.it
Andrea E. Naimoli - andrea.naimoli@unitn.it
Giacomo Colle - giacomo.colle@unitn.it
C.D.L.: Informatica (INF)
A.A.: 2025-2026
Luogo: DISI, Università di Trento
URL: <https://shorturl.at/cxAAS>



Ultimo aggiornamento: 29 ottobre 2025

Terms of Use and Copyright

USE

This material (including video recording) is intended solely for students of the University of Trento registered to the relevant course for the Academic Year 2025-2026.

SELF-STORAGE

Self-storage is permitted only for the students involved in the relevant courses of the University of Trento and only as long as they are registered students. Upon the completion of the studies or their abandonment, the material has to be deleted from all storage systems of the student.

COPYRIGHT

The copyright of all the material is held by the authors. Copying, editing, translation, storage, processing or forwarding of content in databases or other electronic media and systems without written consent of the copyright holders is forbidden. The selling of (parts) of this material is forbidden. Presentation of the material to students not involved in the course is forbidden. The unauthorised reproduction or distribution of individual content or the entire material is not permitted and is punishable by law.

The material (text, figures) in these slides is authored mostly by Roberto Sebastiani, with contributions by Marco Roveri, Alessandro Armando, Enrico Giunchiglia e Sabrina Recla.

- 1 Allocazione e Deallocazione Dinamica
- 2 Array e Stringhe Allocati Dinamicamente
- 3 Array Multidimensionali Allocati Dinamicamente

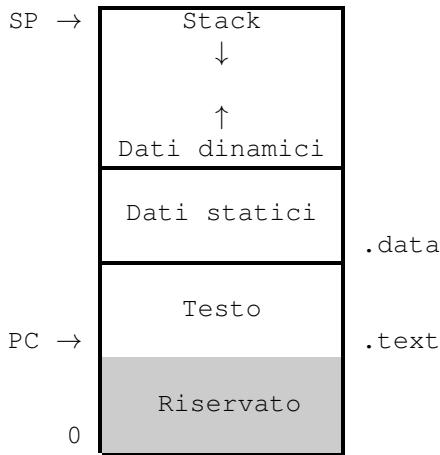
Uso Dinamico della Memoria

- L'allocazione statica obbliga a definire la struttura e la dimensione dei dati a priori (compiler time)
 - Non sempre questo è accettabile e/o conveniente
 - Esempio: dimensione di un array fissa e stabilita a priori (`int a[100];`)
- In C++ è possibile gestire la memoria anche **dinamicamente**, ovvero **durante l'esecuzione del programma**
- Memoria allocata nello **store** (**heap**), un'area esterna allo stack
- L'accesso avviene tramite **puntatori**
- L'allocazione/deallocazione è gestita dagli operatori **new** e **delete**

Modello di gestione della memoria per un programma (ripasso)

Area di memoria allocata ad un'esecuzione di un programma:

- **Area programmi**: destinata a contenere le **istruzioni** (in linguaggio macchina) del programma
- **Area dati statici**: destinata a contenere **variabili globali** o **allocate staticamente** e le **costanti** del programma.
- **Area heap**: destinata a contenere le **variabili dinamiche** (di dimensioni non prevedibili a tempo di compilazione) del programma.
- **Area stack**: destinata a contenere le **variabili locali** e i **parametri formali** delle funzioni del programma.



Allocazione: l'operatore **new**

- Sintassi:

- **new** tipo;
- **new** tipo (valore); (con inizializzazione del valore)
- **new** tipo[dimensione]; (per gli array)

dove **dimensione** può essere un'espressione **variabile** e **valore** deve essere un valore costante di tipo **tipo**

- Esempio:

```
int *p, *q;  
char *stringa;
```

```
p = new int;  
q = new int (5); // Assegna valore 5 all'area di memoria  
stringa = new char[3*i];
```

Allocazione: l'operatore **new** - II

- L'operatore **new/new**[dimensione]:
 1. alloca un'area di memoria adatta a contenere un oggetto (o `dimensione` oggetti) del tipo specificato
 2. la inizializza a `valore` (se specificato)
 3. ritorna l'**indirizzo** (del primo elemento) di tale area
⇒ tipicamente assegnato ad un puntatore

Deallocazione: l'operatore **delete**

- Sintassi:

- **delete** indirizzo;
- **delete**[] indirizzo; (per gli array)

dove il valore dell'espressione `indirizzo` deve essere l'indirizzo di una cella precedentemente allocata da una chiamata a **new**

- Esempio:

```
int *p;  
char *stringa;  
  
p = new int;  
stringa = new char[30];  
delete p;  
delete[] stringa;
```


Deallocazione: l'operatore `delete` - II

- L'operatore `delete/delete []` dealloca l'area di memoria precedentemente allocata a partire dall'indirizzo specificato
 - se l'indirizzo non corrisponde ad una chiamata a `new` \implies errore
 - un'area allocata da `new` [resp. `new [. . .]`] deve essere deallocata con `delete` [resp. `delete []`] (altrimenti comp. non specificato)
- Al termine del programma anche la memoria allocata con `new` viene automaticamente deallocata

Deallocare un'area di memoria:

- significa che quell'area non é più “riservata”
⇒ può essere ri-allocata
- non significa che il suo contenuto venga cancellato!
⇒ valore potenzialmente ancora accessibile per un po' di tempo (**non noto a priori!!!**)
⇒ **facile** non accorgersi di situazione di errore!!!

Esempi: **new** e **delete** su variabili semplici

- Esempio con **new**:
{ ALLOC_DINAMICA/new1.cc }
- variante:
{ ALLOC_DINAMICA/new2.cc }
- ... con inizializzazione:
{ ALLOC_DINAMICA/new3.cc }
- esempio di allocazione e deallocazione:
{ ALLOC_DINAMICA/newdelete1.cc }
- tentativo di deallocazione di variabile statica:
{ ALLOC_DINAMICA/newdelete2.cc }
- indipendenza dal nome del puntatore:
{ ALLOC_DINAMICA/newdelete3.cc }

Durata di un'Allocazione Dinamica

- Un oggetto creato dinamicamente resta allocato finché:
 - non viene esplicitamente deallocato con l'operatore **delete**, oppure
 - il programma non termina
- La memoria allocata con **new** non esplicitamente deallocata con **delete**, può risultare non più disponibile per altri programmi
 - ⇒ spreco di memoria (**memory leak**)
 - ⇒ degrado delle prestazioni della macchina

Regola aurea

In un programma, si deve sempre esplicitamente deallocare tutto quello che si è allocato dinamicamente non appena non serve più.

Gestione dinamica della memoria: pro e contro

- Pro:
 - **Gestione efficiente della memoria**: alloca solo lo spazio necessario
 - Permette la **creazione di strutture dati dinamiche** (liste, alberi, ...)
- Contro:
 - Molto più difficile da gestire
 - Facile introdurre errori e/o memory leaks

Nota

- Esistono strumenti a supporto dell'identificazione di memory leaks:
 - Open source (e.g. `valgrind`, `gperftool`, `-fsanitize=...`)
 - ... e commerciali (e.g. `Parasoft Insure++`, `IBM Rational Purify`)

Allocazione dinamica di Array

- Consente di creare a run-time array di dimensioni diverse a seconda della necessità
- Un array dinamico è un **puntatore** al primo elemento della sequenza di celle

```
int n;  
cin >> n;  
int *a = new int[n]; //allocazione dell'array  
for (int i=0; i<n; i++) {  
    cout << endl << i+1 << ":_";  
    cin >> a[i]; }  
delete[] a;           //deallocazione dell'array
```

- Esempio di cui sopra esteso:
{ ALLOC_DINAMICA/prova.cc }
- Allocazione dinamica array + inizializzazione: non più ammessa:
{ ALLOC_DINAMICA/prova5.cc }

Allocazione dinamica di Stringhe

- Consente di creare a run-time stringhe di dimensioni diverse
- Una stringa dinamica è un puntatore al primo elemento della sequenza di caratteri, terminata da `'\0'`
- L'I/O di una stringa dinamica è gestita automaticamente dagli operatori `>>`, `<<`
- Tutte le primitive su stringhe in `<cstring>` applicano anche alle stringhe dinamiche

```
char * sc, *sb = new char [20];  
cin >> sb;  
sc = new char[strlen(sb)+1];  
strcpy(sc, sb);  
cout << sc;
```

- Esempio di cui sopra esteso:
{ `ALLOC_DINAMICA/prova2.cc` }

Fallimento di **new**

- L'esecuzione di una **new** può **non andare a buon fine** (memoria destinata al programma esaurita)
 - In tal caso lo standard C++ prevede che, se non diversamente specificato, **new** richieda al s.o. di abortire il programma.
- Soluzione: usare "**new (nothrow)**"
 - Con l'opzione "**nothrow**", **new** non abortisce ma restituisce "**NULL**" **in caso di impossibilità ad allocare** la memoria richiesta.
 - Esempio:

```
char *p = new (nothrow) char [mymax] ;  
...  
if (p != NULL) ...
```


Esempi

Suggerimento 1: aprire shell con comando “top” attivo

Suggerimento 2: su `bash` provare ad eseguire con

```
(ulimit -v 5000000; ./a.out)
```

- allocazione eccessiva:
{ `ALLOC_DINAMICA/prova3.cc` }
- ... con deallocazione:
{ `ALLOC_DINAMICA/prova3_bis.cc` }
- deallocazione non dipende dal nome del puntatore!:
{ `ALLOC_DINAMICA/prova3_tris.cc` }
- `delete []` richiede l'indirizzo del primo elemento allocato!:
{ `ALLOC_DINAMICA/prova3_err.cc` }
- uso di `new (nothrow)`:
{ `ALLOC_DINAMICA/prova4_nothrow.cc` }

Restituzione di Array II¹

Una funzione può restituire un array se allocato **dinamicamente** al suo interno.

```
int *times(int a[], ...) {  
    int * b = new int[10];  
    (...)  
    return b;  
}  
  
int v[10] = {1,2,3,4,5,6,7,8,9,10};  
int * w = times(v, ...);
```

- versione non corretta, vedi Ch. 6:
{ ALLOC_DINAMICA/err_restituzione_array2.cc }
- versione corretta:
{ ALLOC_DINAMICA/restituzione_array.cc }

¹Si veda per confronto la slide omonima in Cap. 06.

Responsabilità di allocazione/deallocazione dinamica

Quando si usa allocazione dinamica di un dato (e.g. di un array) che viene passato tra più di una funzione, il programmatore deve:

- decidere **quale funzione ha la responsabilità di allocare il dato**
 - rischio di mancanza di allocazione \implies segmentation fault
 - rischio di allocazioni multiple \implies memory leak
- decidere **quale funzione ha la responsabilità di deallocarlo**
 - rischio di mancanza di deallocazione \implies memory leak
 - rischio di deallocazioni multiple \implies segmentation fault
- adeguare **il passaggio di parametri delle funzioni** in tal senso.
 - rischio di mancanza di allocazione \implies segmentation fault

Nota importante

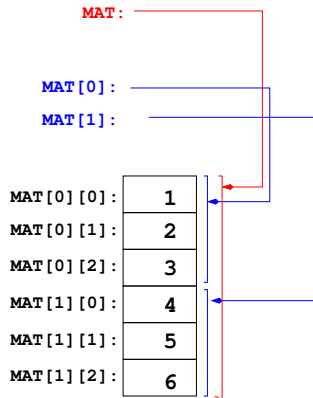
È fondamentale **concordare preventivamente la responsabilità dell'allocazione e deallocazione** quando il codice è sviluppato in team!

Esempi

- nessuna allocazione, passaggio per valore:
{ ALLOC_DINAMICA/responsabilita1_err.cc }
- allocazione esterna alla funzione get, passaggio per valore:
{ ALLOC_DINAMICA/responsabilita1.cc }
- allocazione interna alla funzione get, passaggio per valore:
{ ALLOC_DINAMICA/responsabilita2_err.cc }
- allocazione interna alla funzione get, passaggio per riferimento:
{ ALLOC_DINAMICA/responsabilita2.cc }
- doppia allocazione, passaggio per riferimento:
{ ALLOC_DINAMICA/responsabilita2_memleak.cc }
- deallocazione interna alla funzione print (insensata e pericolosa)!:
{ ALLOC_DINAMICA/responsabilita3.cc }
- deallocazione interna alla funzione print (insensata e pericolosa)!:
{ ALLOC_DINAMICA/responsabilita3_2delete.cc }
- funzione di deallocazione esplicita:
{ ALLOC_DINAMICA/responsabilita4.cc }

Struttura di un array bidimensionale (statico)

```
int MAT[2][3] = {{1,2,3},{4,5,6}};
```



Esempio di allocazione statica (da esempi su “MATRICI”):
{ `ALLOC_DINAMICA/matrix_sta.cc` }

Allocazione dinamica di un array multidimensionale

- In C++ non è possibile allocare direttamente un array multi-dimensionale in modo dinamico

⇒ array multidimensionali e puntatori sono oggetti **incompatibili**.

```
int * MAT1 = new int[2][3]; // ERRORE
```

```
int ** MAT2 = new int[2][3]; // ERRORE
```

- “`new int[2][3]`” restituisce l'indirizzo di 2 oggetti consecutivi di tipo “`int[3]`”, incompatibili sia con “`int *`” che con “`int **`”

Esempio di cui sopra, espanso:

```
{ ALLOC_DINAMICA/matrix_din_err.cc }
```

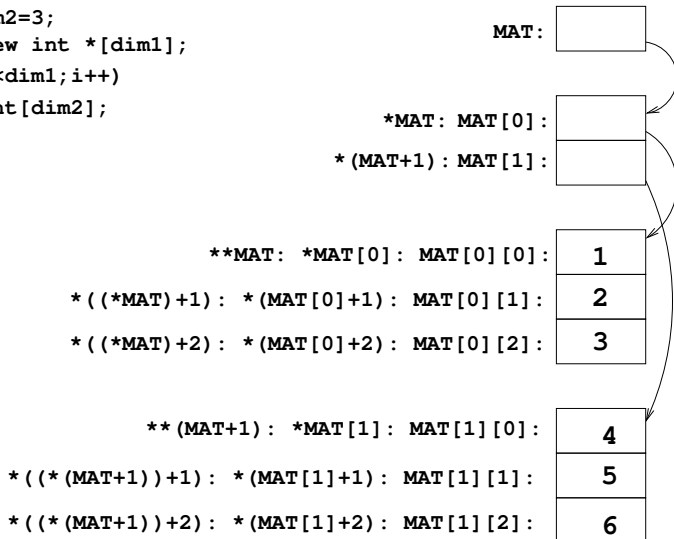
Array dinamici multidimensionali

- In C++ si possono definire array dinamici multidimensionali come **array dinamici di array dinamici ...**
- Tipo base: puntatore di puntatore ...
- Gli operatori “[]” funzionano come nel caso statico
 - `MAT[i]` equivalente a `* (MAT+i)`,
 - `MAT[i][j]` equivalente a `* ((* (MAT+i)) +j)`,
- L'allocazione richiede un ciclo (o più)

```
int ** M;           // puntatore a puntatori a int
M = new int *[dim1]; // array dinamico di puntatori
for (int i=0; i<dim1; i++)
    M[i] = new int[dim2]; // allocazione di ciascun array
```

Struttura di un array bidimensionale (dinamico)

```
int dim1=2, dim2=3;  
int ** MAT = new int *[dim1];  
for (int i=0;i<dim1;i++)  
    M[i] = new int[dim2];
```



Esempio di cui sopra, espanso:



Esempi: allocazione e gestione di matrici dinamiche

- Operazioni matriciali su matrice dinamica:
{ ALLOC_DINAMICA/matrix.cc }
- idem, con il nuovo tipo “matrix” (uso di **typedef**):
{ ALLOC_DINAMICA/matrix_typedef.cc }
- come sopra, con unica funzione di allocazione matrice:
{ ALLOC_DINAMICA/matrix_v2_typedef.cc }

Esempi: deallocazione di matrici dinamiche

Suggerimento 1: aprire shell con comando “top” attivo

Suggerimento 2: su `bash` provare ad eseguire con

```
(ulimit -v 5000000; ./a.out)
```

- allocazione senza deallocazione di matrici dinamiche:
{ `ALLOC_DINAMICA/matrix2.cc` }
- .. con deallocazione mediante `delete[]`:
{ `ALLOC_DINAMICA/matrix3.cc` }
- ... con deallocazione completa di matrici:
{ `ALLOC_DINAMICA/matrix4.cc` }

⇒ Anche la deallocazione richiede un ciclo (o più)

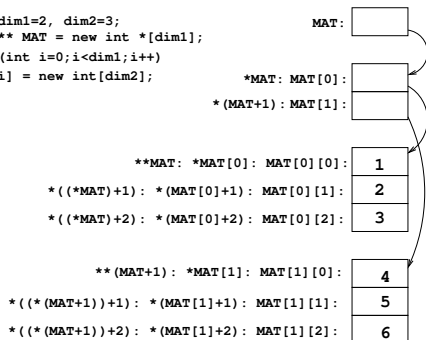
Array bidimensionali dinamici vs. statici

Sebbene concettualmente simili, gli array multidimensionali dinamici e statici sono sintatticamente oggetti diversi e non compatibili

(uno è un “`int **`”, l'altro un “`int * const *`”)

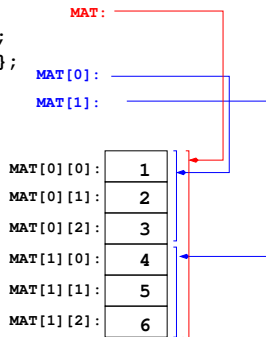
Array bidimensionale dinamico

```
int dim1=2, dim2=3;  
int ** MAT = new int *[dim1];  
for (int i=0; i<dim1; i++)  
    M[i] = new int[dim2];
```



Array bidimensionale statico

```
int MAT[2][3] = ;  
{ {1, 2, 3}, {4, 5, 6} };
```



Array bidimensionali dinamici vs. statici II

```
void print_matrix_dim(float ** a, ...) {...}  
void print_matrix_sta(float a[][d2a], ...) {...}  
  
float A[d1a][d2a] = {{1,2,3},{4,5,6}};  
float ** B;  
B = read_matrix(d1b,d2b);  
// B = A;                                // errore  
// print_matrix_dim (A, d1a, d2a); // errore  
print_matrix_dim(B, d1b, d2b);  
print_matrix_sta(A, d1a, d2a);  
// print_matrix_sta(B, d1b, d2b); // errore
```

Esempio di cui sopra, espanso:

```
{ ALLOC_DINAMICA/matrix_stavsdin.cc }
```