40. Gestione dinamica della Memoria
L'allocazione statica obbliga a definire la struttura e la dimensione dei dati a priori (a compiler time, tempo di compilazione). 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. In questo caso la memoria è allocata nello store (*heap*), un'area esterna allo stack, di dimensione potenzialmente infinita (in una macchina a 64 bit, 2^64 celle di memoria).

L'accesso avviene solamente tramite i **puntatori** e l'allocazione e la deallocazione è gestita dagli operatori ***new*** e ***delete***.

40.1 Modello di gestione della memoria per un programma
Area della memoria destinata ad un'esecuzione di un programma si divide in:

- Area programmi: destinata a contenere le istruzioni (in linguaggio macchina) del programma. In codice assembly, che fornisce informazioni su come eseguire il 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.

Stack ha indirizzo più alto della heap.

41. 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;  //rappresenta un array di carattere

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

Un modo più compatto per allocare memoria è questo: ***int *p = new int(val);***

Quando si esegue l'istruzione *p = new int;* viene allocata nella heap una parte di memoria atta a contenere un intero. In *p* c'è il puntatore a quell'area della memoria, se faccio dereference di *p* accedo a ciò che c'è nell'area di memoria.

In stringa c'è un puntatore al primo elemento dell'array. Poi si possono usare i puntatori come array.

L'operatore *new* e *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).

Per maneggiare l'area di memoria dinamica uso quindi la dereference del puntatore che contiene l'indirizzo di quella particolare area di memoria. Uso quindi *(*p)* e accedo all'area di memoria. Posso anche salvare l'indirizzo di quell'area di memoria in un altro puntatore e maneggiarla facendo la dereference di esso.

42. Deallocazione: l'operatore *delete*

Sintassi
delete indirizzo;
delete[] indirizzo; (per gli array) //[] indica di non deallocare solo un elemento ma tutta l'area di memoria dell'array
dove il valore dell'espressione *indirizzo* deve essere l'indirizzo della celletta precedentemente allocata dalla chiamata *new*.

Esempio
p = new int;
stringa = new char[30];

delete p;
delete[] stringa;

L'operatore *delete* e *delete[]* dealloca l'area di memoria precedentemente allocata a partire dall'indirizzo specificato:

- se all'indirizzo non corrisponde ad una chiamata *new* dà un errore
- un'area allocata da *new* deve essere deallocata con *delete*

Al termine del programma anche la memoria allocata con *new* viene automaticamente dealllocata.

Note sulla deallocazione
Deallocare un'area di memoria significa che quell'area non è più "riservata" e può essere riallocata.

Importante: Non significa che il suo contenuto venga cancellato: il suo valore è potenzialmente ancora accessibile per un po' di tempo (non noto a priori). Il puntatore a quell'indirizzo è sempre valido.
Quando poi quell'area di memoria viene deallocata, non si può più accedere a quell'area.
Dopo la deallocazione è bene non accedere più a quell'area di memoria. Il compilatore me lo permette però vado ad accedere ad aree di memoria che non sono più in mio controllo. Infatti a runtime esce il messaggio di errore *free()* seguito da qualcosa e il programma abortisce. *free()* vuol dire che stiamo deallocando memoria. Uno dei possibili messaggi è *invalid pointer*, se facciamo il *delete* su un puntatore, ad esempio, ad una cella dello stack, ovvero con variabili statiche.

!N.B.! La cosa importante nella deallocazione è che la *delete* sia fatta sull'**indirizzo di memoria** precedentemente allocata. Ciò significa che se il puntatore che prima puntava a quella data area di memoria ha cambiato valore (cioè punta un'altra area di memoria) non dobbiamo fare la *delete* su quel puntatore, ma sul puntatore che contiene l'indirizzo di memoria precedente allocato.

! Vedere l-value del puntatore su cui si fa la *delete* !

43. Durata di un'allocazione dinamica
Un oggetto creato dinamicamente resta allocato finché:

- non viene esplicitamente deallocato con l'operatore delete;
- il programma non termina.

La memoria allocata con *new* non esplicitamente deallocata con *delete*, può risultare non più disponibile per altri programmi (che stanno girando sul computer), causando uno spreco di memoria (memory leak) e/o un degrado delle prestazioni della macchina (si fa ciò che viene chiamato *swap*: passare dalla memoria fisica a quella su disco).
Inoltre è importante deallocare memoria perché se la heap si satura non è più possibile fare altre allocazioni.

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

43.1 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 dei memory leaks sia *Open Source* (come *valgrind*, *gperftool* (fa detection di memory leaks e analisi di efficienza del programma), *-fasnitize=*…) che commerciali (*Parasoft Insure++*, *IBM Rational Purify*).

Il comando **valgrind** fornisce informazioni importanti e segnala errori quando ad esempio facciamo accesso ad un area di memoria precedente deallocata.
Si lancia facendo *valgrind ./a.out*.

- *heap summary* (total heap usage)
- ***leak summary*** (mi dice se ci sono stati dei memory leaks, ovvero se non ho deallocato la memoria non utilizzata alla fine del programma. Se ho deallocato tutto non ci sarà la parte *leak summary* e dopo la parte *heap summary* ci sarà la scritta "All heap blocks were freed -- no leaks are possible").
- *error summary* se facciamo errori come ad esempio accedere ad aree di memoria deallocate. L'*error summary* si trova in fondo, più sopra c'è la spiegazione di dove si trova l'errore.

Se lo lanciamo con la flag *-s*, ovvero *valgrind -s ./a.out* sotto il *leak summary* (o equivalente) ci sarà l'*error summary* con una spiegazione più dettagliata degli errori.

44. Allocazione dinamica di Array
Consente di creare a run-time array di dimensioni diverse a seconda delle necessità. Un array dinamico è un puntatore al primo elemento della sequenza di celle. Come abbiamo visto, un puntatore può essere trattato come un array, quindi si può scorrere un array di dimensione *dim*, inserita dall'utente, in questi due modi:

```
for(int i=0;i<dim;i++)
  cout << *(p+i) << endl;
  cout << p[i] << endl;
```

Ricordiamo allocazione e deallocazione di array.

```
int n;
cout << "Quanti elementi nel tuo array? " << endl;
cin >> n;
int *a = new int[n];
for(int i=0;i<n;i++)
  cin >> a[i];
delete[] a;      //si fa così la deallocazione di un array
```

! È importante fare la deallocazione non appena non uso più la variabile **!**

Importante: non è più consentita inizializzazione in fase di definizione, come per gli array statici.
int a = new int[n] = {1,2,3};* //non è consentito

44.05 Array dinamici e funzioni
Diversamente dagli array statici, gli array dinamici **NON** vengono passati per riferimento alla funzione. Senza specifcare niente vengono passati per valore (infatti sono dei puntatori).
*… funz(**tipo*&**);* in questo modo si passa un array dinamico (cioè un puntatore) alla funzione per riferimento. Passando un array dinamico per valore, se il *new* viene fatta nella funzione e il *delete* nel main, il *delete* sarà errato perché corrisponderà un indirizzo diverso.

44.1 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, chiaramente terminata da '/0'.
L'I/O è gestita automaticamente dagli operatori >> e <<, come per le stringhe statiche.
Tutte le primitive su stringe in <*cstring*> applicano anche alle stringhe dinamiche.

```
int dim;
cout << "Quante lettere ci sono nella tua parola? " << endl;
cin >> dim;
dim++; //la stringa ha dimensione numero_lettere+1 perché deve contenere anche '/0'

char* sc, *sb = new char[20]; //stringa di massimo 19 caratteri
cin >> sb;

sc=new char[strlen(sb)+1];
strcpy(sc,sb);
cout << sc;

delete[] sb;
delete[] sc;
```

Per manipolare stringhe e array si usa il fatto che un puntatore può essere usato come un array, quindi per stamparne il contenuto stampo *sc*, *sb*, *a*, … (non faccio la dereference è più comodo usare i puntatori come array, cioè associandogli []).

Alcuni pattern importanti

Come faccio a far scrivere all'utente una parola e salvarla in una stringa con esattamente i caratteri di quella parola (+1) senza chiedergli i caratteri totali della parola? Faccio una stringa abbastanza grande poi salvo la parola in quella stringa e usando *strlen* alloco una stringa di quella dimensione (+1) e copio la stringa iniziale nell'altra. È bene farlo con due stringhe dinamiche, cosicché si deallochi immediatamente l'area di memoria della prima stringa, ormai inutile.

```
char* sc, *sb = new char[50];
cin >> sb;

sc=new char[strlen(sb)+1];      //lunghezza esattamente della stringa+1
strcpy(sc,sb);                  //si memorizza la stringa
delete[] sb;                    //ora non ci serve più la prima stringa
… ;
delete[] sc;                    //deallocazione della seconda quando non ci serve più
```

Faccio la stessa cosa se voglio creare una stringa concatenandone 2.

```
char* sa = new char[50];
char *sb = new char[50];
char* sc;

cin >> sa >> sb;
sc = new char[strlen(sa)+strlen(sb)+1];
strcat(sc,sa);
strcat(sc,sb);
```

44.2 Fallimento di *new*

L'esecuzione di una *new* può non andare a buon fine (ad esempio memoria destinata al programma esaurita).
In tal caso lo standard C++ prevede che, se non diversamente specificato, *new* richieda al sistema operativo 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.

*char *p = new (nothrow) char[mymax];*

if (p!=NULL){
//operazione ha avuto successo e posso usare l'area di memoria
}

if(p==NULL){
cerr << "Operazione fallita. Heap satura" << endl; exit(0); //per uscire in modo controllato
//oppure
delete … ; //eliminiano parti di memoria inutili e proviamo a rifare il *new*
*char *p = new(nothrow) char[mymax];*
}

ulimit
(ulimit -v 50000; ./a.out), dove *-v* indica la memoria virtuale che il programma può usare. Di norma è unlimited, ma se specifico come sopra limito la memoria virtuale a 50 000 kB (= 50 MB).

*char *p = new char [1024*1024*1024];* //allocazione dinamica di un array da 1 GB

N.B! È molto importante per evitare questo tipo di problemi DEALLOCARE la memoria quando non la usiamo più (soprattutto nei cicli), ricordando che è importante l'indirizzo che si dealloca e non il puntatore che lo contiene.

45. Restituzione di Array
Ricordiamo che una funzione può restituire un'array statico solo se definito globalmente o nel main. Se allocato dinamicamente al suo interno, una funzione può restituire un array (in un altro array dinamico).

*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,...);*

! *w* diventa un puntatore ad un area di memoria nell'heap. Quindi è necessario effettuare *delete[] w;* quando non lo usiamo più.

Chiamando la funzione un'altra volta su un altro puntatore, si restituisce un diverso indirizzo di memoria. Ciò significa che si creano *n* diversi puntatori ad aree heap per *n* chiamate della funzione. **Bisognerà fare *n* delete[]!**

45.1 Responsabilità della allocazione e della deallocazione dinamica
Quando si usa allocazione dinamica di un dato (ad esempio 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 : segmentation fault
  - rischio di allocazioni multiple: memory leak
- decidere quale funzione ha la responsabilità di deallocarlo
  - rischio di mancanza di deallocazione: memory leak
  - rischio di deallocazioni multiple: segmentation fault
- adeguare il passaggio di parametri delle funzioni in tal senso.
  - rischio di mancanza di allocazione: segmentation fault.

N.B! L'area di memoria è visibile a tutte le funzioni del programma! Quindi basta fare una allocazione! **Tenere traccia dei puntatori e delle aree di memoria allocate!**

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

Note su allocazioni e deallocazioni
Quando un'area di memoria viene allocata, se ho il puntatore a quell'indirizzo, posso accedere da qualunque funzione del mio programma a quell'area di memoria. Quindi dopo aver allocato un'area di memoria, la posso deallocare in qualunque funzione.

Importante: ad *n* allocazioni di memoria corrispondono *n* deallocazioni (non importa dove ne con che puntatore, ciò che conta è fare il *delete* dell'indirizzo di memoria precedentemente allocato).
Ciò però non vuol dire che se fisicamente nel mio codice leggo 3 *new* devo effettuare 3 *delete*. Se magari un *new* è in una funzione che viene richiamata due volte nel mio codice c'è un solo *new* ma io devo mettere due *delete*.

! Attenzione a passare puntatori alle funzioni per riferimento e per valore e a fare allocazioni **!**

Attenzione a non allocare aree di memoria sovrascrivendo il puntatore che presentava un indirizzo di un'area di memoria precedentemente allocata. L'area di memoria allocata all'inizio è persa definitivamente, perché l'indirizzo di memoria a cui puntava il puntatore è andato perso, sovrascritto dall'indirizzo di un'altra area di memoria.

Aiuto per allocazioni e deallocazioni: dopo la compilazione lancio valgrind con i seguenti flag *valgrind --leak -check=full ./a.out* e mi mostra dove la memoria è stata allocata e dove è stata sovrascritta.

Una cosa per evitare problemi in allocazione e deallocazione è la seguente:

- definire una funzione *alloc* in cui viene passato per riferimento un puntatore e fa tutte le allocazioni del caso (*void alloc(int*&)* oppure *tipo* alloc(…);* dove al posto … metto come parametri informazioni utili alla allocazione come dimensioni, nel caso di array).
- definire una funzione *dealloc* che si occupa di deallocare l'area di memoria, il puntatore qua è passato per valore cosicché nella funzione chiamante esso punta ancora all'area di memoria (però è consigliabile non usarlo più o comunque usarlo per allocare una diversa area di memoria).
    - Il puntatore può essere anche passato per riferimento, effettuare la deallocazione e poi scrivere *p=NULL;* (*void dealloc(int*&)*), in modo da rendere non più valido l'indirizzo di memoria di questo puntatore.

46. 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 (sia con tipo* che con tipo**).**
*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]*".

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.
Con gli array dinamici multidimensionali:
MAT[i] equivalente a **(MAT+i)*,
MAT[i][j] equivalente a **((*(MAT+i))+j)*,

46.1 Allocazione e deallocazione
L'allocazione richiede un ciclo (o più).
Sintassi
```
tipo ** matrix = new tipo *[dim1];
for(int i=0;i<dim1;i++){
matrix[i] = new tipo[dim2];
}
```

Esempio
```
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
```

Uguale per la deallocazione
```
for(int i=0; i<dim1; i++)
  delete[] M[i];

 delete[] M;    //è importante deallocare questa per ULTIMA
```

N.B.! Notare che in fase di allocazione faccio dim1+1 *new* a cui corrispondono dim1+1 *delete*. Tutto ciò che alloco viene deallocato.

Per la manipolazione uso sempre la notazione di subscripting o l'aritmetica dei puntatori come mostrato sopra. Tutto uguale agli array statici.

46.2 Matrici dinamiche e funzioni
Per restituire una matrice dinamicamente
*** tipo funzione(…);*

Chiamata: *tipo ** m = funzione(…);*
Con le matrici dinamiche posso creare la matrice dinamicamente all'interno della funzione e restituirla.

Ricordarsi di deallocare e di non sovrascrivere matrici (altrimenti la memoria diventa non più deallocabile).

Per passare come parametro una matrice dinamica
*tipo funzione(tipo **);*
Per passarne una statica devo fare invece *tipo funzione(tipo [][DIM])* specificando la seconda dimensione (tutte quelle successive alla prima).

46.3 L'operazione di *typedef*

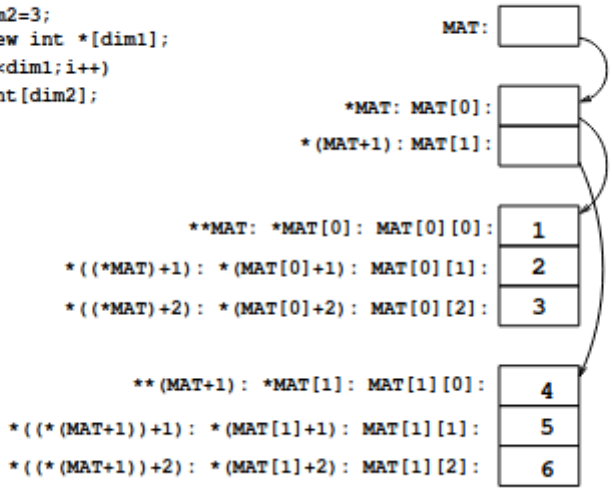
Onde evitare di scrivere ogni volta un tipo, se questo è lungo e complesso, si può usare il *typedef*.

Ad esempio se devo creare degli array multidimensionale e non ho sbatti di scrivere *tipo***, posso, fuori dal main (nello spazio globale), scrivere:
typdef (tipo da sostiture) (id sostituente);

Ad esempio posso scrivere *typedef float** *matrix;* così da poter definire delle matrici dinamiche semplicemente scrivendo *matrix m* anziché *float** *m*.

46.2 Struttura di un array bidimensionale dinamico e statico

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



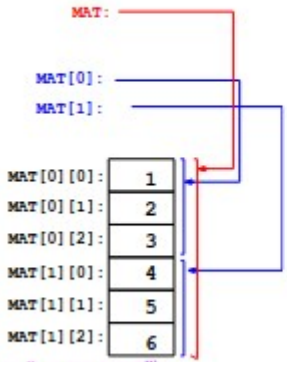
MAT è un array di puntatori (nello specifico MAT punta alla sua prima cella, ovvero MAT[0])
MAT[0] è un array di interi, ovvero contiene un puntatore che punta a MAT[0] [0] (il primo suo elemento).

Non c'è un layout lineare, ma funziona comunque l'aritmetica dei puntatori.

Statico:

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

Layout lineare rispetto a quello degli array dinamici.



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 ****).