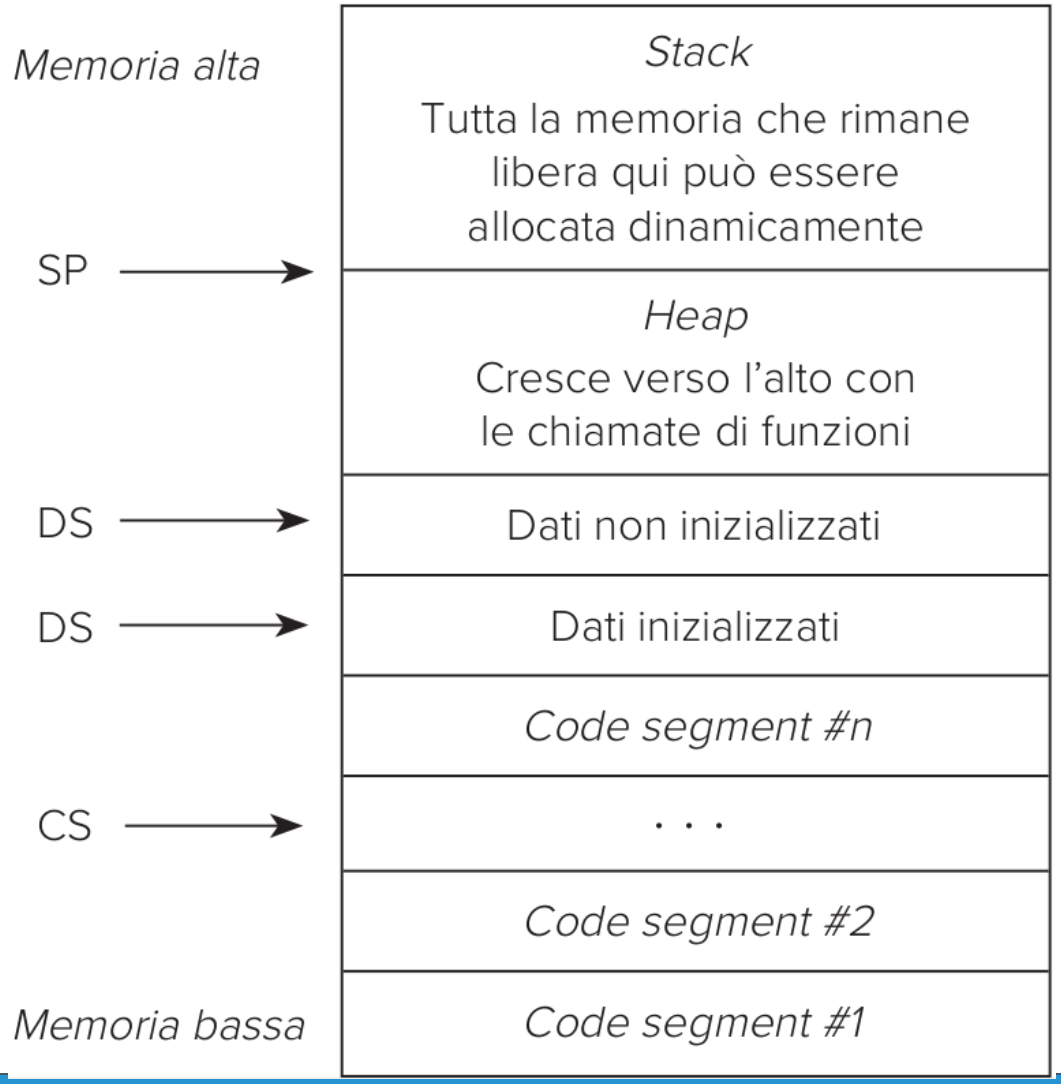
Capitolo 8

Allocazione dinamica della memoria

pp. 221-233

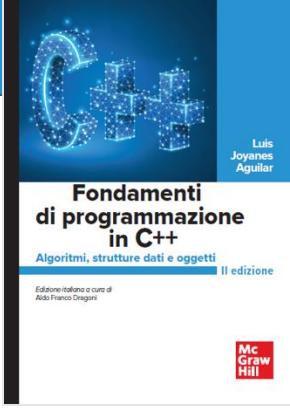
Presenta: Prof Misael Mongiovì

mappa della memoria



E' abbastanza frequente non conoscere la quantità di memoria necessaria fino al momento di esecuzione di un programma.
Per risolvere questo problema si ricorre ai puntatori ed alla allocazione dinamica della memoria.
Il sistema operativo assegna al programma tre specifici segmenti della memoria principale:

Il code segment: che ospita il codice stesso
Il data segment: che ospita costanti e variabili globali
Lo stack: che a runtime conterrà le variabili locali alle funzioni, i parametri delle funzioni e le copie dei registri per restituire il controllo alle funzioni chiamanti al termine delle funzioni chiamate. Quindi questa porzione di memoria allocata in questo segmento cresce e decresce continuamente durante l'esecuzione del programma.
La memoria che rimane libera nello stack viene denominata heap e cresce verso l'alto e cresce e decresce dinamcamente, mentre invece lo stack cresce verso il basso.



memoria dinamica: `new`

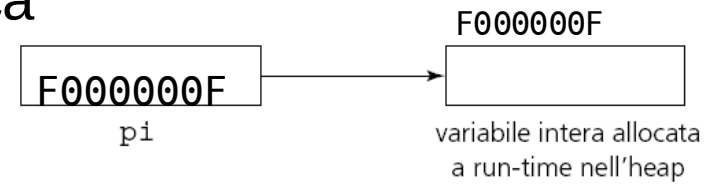
- *tipo* puntatore* = **new** *tipo* // non array
- *tipo* puntatore* = **new** *tipo*[dimensione] // array

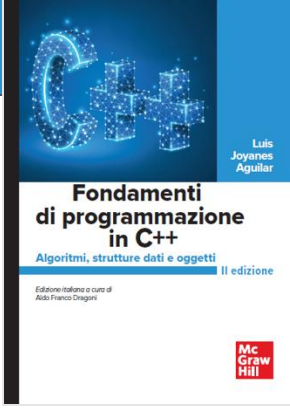
- genera dinamicamente una variabile di un certo tipo assegnandole un blocco di memoria della dimensione opportuna
- restituisce un puntatore che contiene l'indirizzo del blocco di memoria allocato, cioè della variabile, la quale sarà quindi accessibile dereferenziando il puntatore

```
char* p = new char[100]; // genera dinamicamente un
                         // vettore di cento char
```

- se il puntatore è già stato definito si può semplicemente usare `new` per assegnargli la variabile dinamica

```
int* pi;
pi = new int;
```





memoria dinamica: `delete`

```
delete puntatore// non array
delete [] puntatore  // per array
```

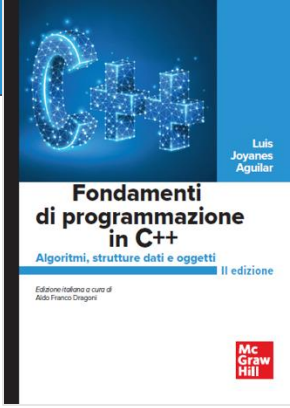
- libera la memoria allocata dinamicamente perché possa eventualmente essere riallocata mediante successive chiamate all'operatore `new`
- lo spazio assegnato per le variabili dinamiche:

```
int* ad = new int;
char* adc = new char[100];
```

si può liberare con le istruzioni:

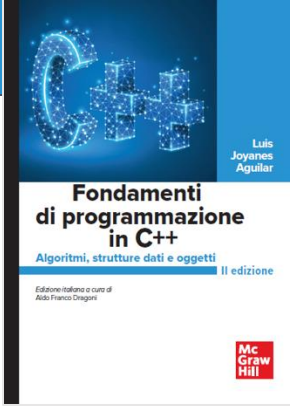
```
delete ad;
delete [] adc;
```

- rende riutilizzabile la memoria puntata ma non cancella il puntatore che può quindi essere riutilizzato, ad esempio per puntare un'altra variabile successivamente allocata con `new`



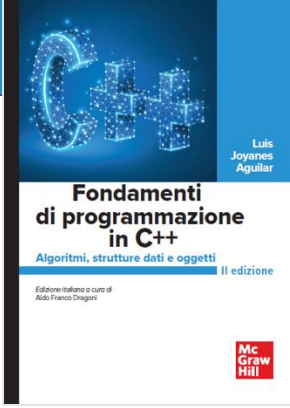
esempio

```
#include <iostream>
#include <string>
int main()
{
    int lunghezza_stringa;
    char *p;
    cout << "Quanti caratteri si assegnano?";
    cin >> lunghezza_stringa;
    cin.ignore(1);
    p = new char[lunghezza_stringa+1];
    strncpy(p, "Carchel tambien...", lunghezza_stringa);
    p[lunghezza_stringa]='\0'
    cout << p << endl;
    delete p;
    cout << "Prema Invio per continuare";
    cin.get();
}
```



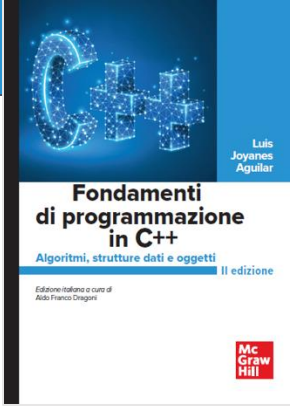
esempio

```
#include <iostream>
#include <cstring>    //contiene la funzione strlen
using namespace std;
int main()
{
    char* str = "Mi pueblo es Carchelejo en Jaen";
    int lung = strlen(str); //ottiene la lunghezza di str
    char* p = new char[lung+1]; // memoria a str + '\0'
    strcpy (p, str);            // copia str a p
    cout << "p = " << p << endl; // p sta ora in str
    delete [] p;
    return 0;
}
```



esempio

```
#include <iostream>
#include <stdlib>
using namespace std;
int main()
{
    cout << "Quanti elementi ha il vettore?";
    int lun;
    cin >> lun;
    int* mioArray = new int[lun];
    for (int n = 0; n < lun; n++)
       mioArray[n] = n + 1;
    for (int n = 0; n < lun; n++)
       cout << '\n' << mioArray[n];
    delete [] mioArray;
    return 0;
}
```



memoria dinamica e puntatori

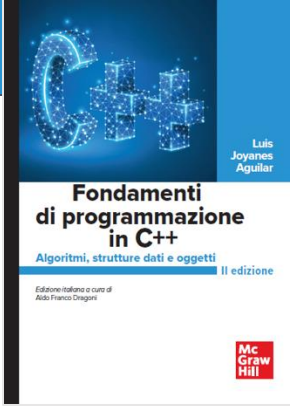
- senza la memoria dinamica la cosa sarebbe un po' sciocca

```
studente a, b;
a.nome[0] = 'U';
a.nome[1] = 'g';
a.nome[2] = 'o';
a.nome[4] = '\0';
a.data_nascita.gg = 22;
```

-
```
studente* p = &a
(*p).puntatore_al_successivo = &b;
```

- questa operazione comune va scritta in maniera più comoda

```
inizio->puntatore_al_successivo = &b;
```



memoria dinamica e puntatori

- un puntatore in una struttura può essere usato per collegare fra loro strutture generate dinamicamente per costruire “liste semplici”

```
struct elem
{ <qualunque tipo> info;
  elem* puntatore_al_successivo;
}
```

- con più puntatori si possono realizzare “liste doppiamente concatenate” (2) o “alberi” (2 o più puntatori)