

UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

**Allocare la memoria usando il free store**

Stefano Ghidoni



DIPARTIMENTO  
DI INGEGNERIA  
DELL'INFORMAZIONE



# Agenda

- Come creare std::vector?
- Acquisizione di memoria dal free store
- Puntatori e free store
- Garbage

# Creare std::vector



# Richiamo: vector

- Abbiamo già usato gli `std::vector`
  - Sequenza di oggetti uguali
  - Riferimento agli elementi: []
  - Aggiunta di un elemento: `push_back()`
  - Numero di elementi: `size()`
  - Accesso con verifica (se richiesto)
- Altri container utili:
  - `std::string`
  - `std::list`
  - `std::map`
  - ...



## std::vector

- std::vector ha un comportamento intelligente
  - Cambia dimensione per ospitare gli elementi che inseriamo
  - Conosce la propria dimensione
- Questo comportamento non è supportato in HW – una sovrastruttura



# Struttura dinamica del vector

- std::vector è una struttura dinamica
  - Diverso dagli array, che hanno dimensione fissa
    - Dovrei poterne modificare la dimensione
  - Non posso usare una struttura statica come una classe:

```
class vector {  
    int size, age0, age1, age2, age3;  
};
```



# Struttura dinamica del vector

- Per poter realizzare una struttura dinamica abbiamo bisogno di poter creare nuovi vettori le cui dimensioni sono note a tempo di esecuzione
- Esiste una zona di memoria che può essere utilizzata in questo modo: il **free store**
- Esiste uno strumento a **basso livello** che ci permette di utilizzare il free store: **allocazione dinamica della memoria**



## std::vector

- Usando il free store gli strumenti a disposizione sono molto meno evoluti:
  - Memoria
  - Indirizzi di memoria
- Il comportamento intelligente di std::vector è **implementato nella libreria standard!**



# Implementare vector

- Come possiamo implementare un vector di double con le stesse caratteristiche di std::vector?
  - Un compito con cui impariamo la gestione della memoria



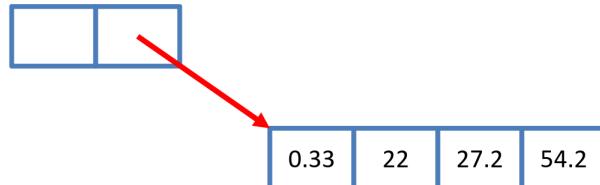
# Struttura di un vector

- Consideriamo un uso semplice:

```
vector<double> age(4);  
  
age[0] = 0.33;  
age[1] = 22;  
age[2] = 27.2;  
age[3] = 54.2;
```

- Da realizzare con questa struttura:

age:

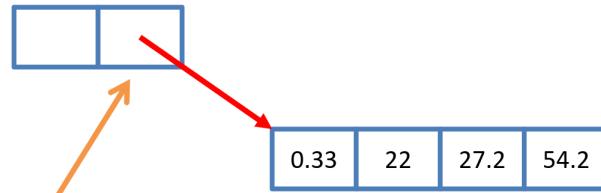


- Come si implementa?
- Come gestire il numero variabile di elementi?



# Struttura di un vector

age:



Come implementare questo  
elemento?

- Come si implementa?
- Come gestire il numero variabile di elementi?



- Il nostro vector può essere creato così:

```
class vector {  
    int sz;  
    double *elem;  
public:  
    vector(int s);          // crea un array di s double e fa sì  
                           // che elem punti ad esso  
    int size() const { return sz; }  
};
```

- Il puntatore serve per gestire un buffer



# Gestione della memoria dei vector

age:



Ora sappiamo gestire questo



Impariamo a gestire questo

- Come si implementa?
- Come gestire il numero variabile di elementi?

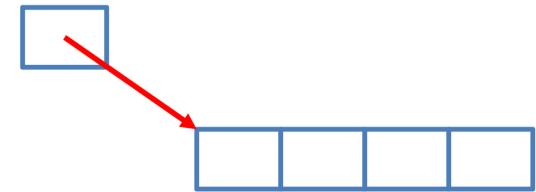


# Allocazione dinamica

- Come accedere al free store?
  - Usando new e i puntatori

```
double* p = new double[4];
```

- new usa il free store
- Questo procedimento prende il nome di **allocazione dinamica della memoria**



Quando creo un oggetto non è obbligatorio mettere il `*new*`. Con "new" si fanno molti procedimenti non utili (viene interpellata la memoria per richiedere un blocco per allocare, ...). Viene usata un'allocazione dinamica della memoria inutile, basterebbe usare lo stack.

Abusare di new è scorretto, infatti triggerà l'allocazione dinamica della memoria.  
In Java è necessario usare new, mentre in cpp se venisse usato sempre sarebbe bad design.



## Puntatori e FS

- new ritorna elementi singoli o array
- Un puntatore **non sa** quanti oggetti sono presenti nell'array ritornato da new
- new ritorna un puntatore anche se allochiamo un solo elemento
  - Ritorna il puntatore all'elemento allocato o al primo elemento se è allocato un array
- La memoria ritornata da new **non ha un nome**
  - Memoria gestita tramite il puntatore restituito

Non sa cosa sia tra i due, deve saperlo il programmatore



- Esempi

```
int* pi = new int;  
int* qi = new int[4];
```

```
double* pd = new double;  
double* qd = new double[n],
```

Può essere una variabile

pi:



qi:

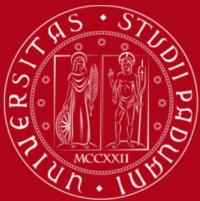


pd:



qd:

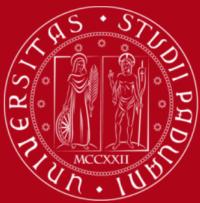




# Accesso agli elementi

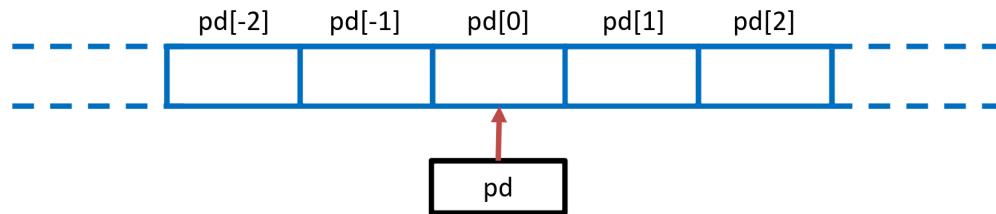
```
double* p = new double[4];
double x = *p;           // primo oggetto puntato
double y = p[2];         // terzo oggetto
```

- Nella gestione della memoria dinamica usiamo spesso:
  - Il nesso tra puntatori e array
  - L'aritmetica dei puntatori
- Ricoridamo che:
  - $*p$  e  $p[0]$  sono equivalenti
  - Applicare  $[]$  a un puntatore equivale a vedere la memoria come un array



# Range e range check

- A questo livello non esiste controllo sui range!
  - Simile agli array stile C



```
pd[2] = 2.2;           // ok!
pd[4] = 4.4;           // ok!
pd[-3] = -3.3;         // ok, ma...
```

"Il mio programma termina misteriosamente..."

- Grave problema
- Esecuzioni diverse possono dare sintomi diversi
- Bug molto difficili da gestire!



# Inizializzazione

- Esempi di inizializzazione del puntatore e della memoria allocata

```
double* p0;           // grosso problema!  
  
double* p1 = new double;    // double non inizializzato  
  
double* p2 = new double {5.5}; // double inizializzato  
  
double* p3 = new double[5];  // array non inizializzato
```



# Inizializzazione

- La memoria allocata con new non è inizializzata per i tipi built-in
- Per ovviare:

```
double* p4 = new double[5] {0, 1, 2, 3, 4}; // array init

double* p5 = new double[] {0, 1, 2, 3, 4}; // dimensione
                                            // dedotta
```



# Inizializzazione con UDT

```
X* px1 = new X;  
X* px2 = new X[17];
```

Chiamata al costruttore di default

- Se il costruttore di default non esiste (es., esiste solo un costruttore che accetta un int):

```
Y* py1 = new Y;          // errore  
Y* py2 = new Y[17];     // errore  
Y* py3 = new Y{13};     // ok  
Y* py4 = new Y[17] {0, 2, 3, 4, ..., 16}; // ok
```



# Controllo di validità

```
if (p0 != nullptr) /* ... */  
  
if (p0) /* ... */
```

- Attenzione quando
  - Il puntatore non è mai stato inizializzato
    - Non è inizializzato a nullptr automaticamente!
  - Il puntatore si riferisce a memoria non valida (liberata)



# Spostamento di un puntatore

- Un puntatore può essere spostato da un oggetto a un altro
  - Differenza con le reference!

```
double* p = new double;
double* q = new double[1000];

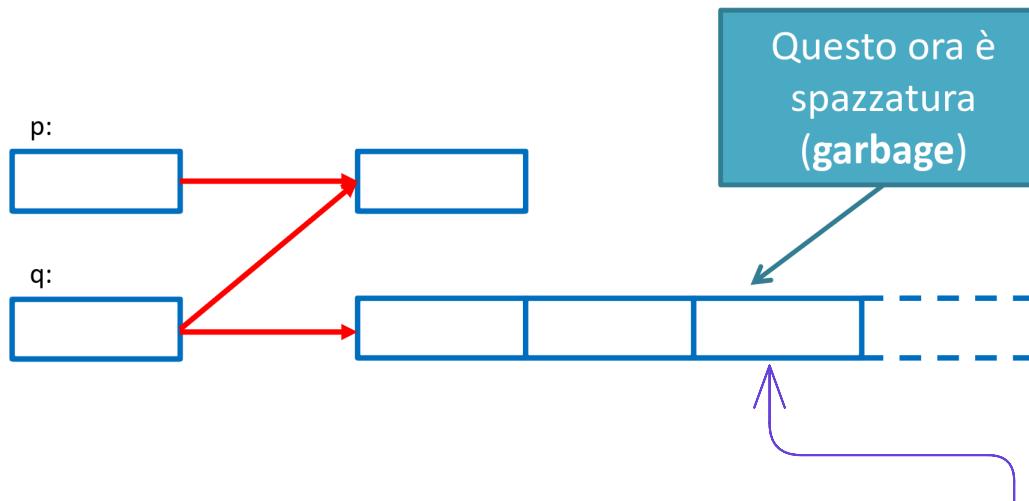
q[700] = 7.7;           // ok
q = p;                 // attenzione!!
double d = q[700];     // attenzione!!
```

- Questo può generare grossi problemi...



# Spostamento di un puntatore

```
double* p = new double;  
double* q = new double[1000];  
  
q = p; // attenzione!!
```



"Nessuno" sa chi è, quindi nessuno può accedervi e quindi neanche cancellarlo, ma la memoria viene occupata



# Garbage

- Un'area di memoria di cui si perdono i riferimenti diventa **garbage**
- Tale zona di memoria rimane allocata ma è irraggiungibile e inservibile
- Questo fenomeno è noto come **memory leak**
  - Può portare all'esaurimento della memoria



# Recap

- Creazione di strutture dinamiche come std::vector
- Allocazione dinamica della memoria
  - Accesso al free store
- Accesso agli elementi e range check
- Inizializzazione
- Garbage



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

## Allocare la memoria usando il free store

Stefano Ghidoni



DIPARTIMENTO  
DI INGEGNERIA  
DELL'INFORMAZIONE