

strutture dinamiche
i vector

I vector

- La libreria standard del C++ mette a disposizione diversi *contenitori*, tra cui il `vector`, che ci permette di realizzare sequenze omogenee di elementi in modo molto efficiente
- il `vector` implementa i meccanismi appena discussi in un contesto molto controllato e molto più facile da usare
- i `vector` sono un esempio di *classe template* — un costrutto avanzato del C++ che non sarà sviluppato ulteriormente in questo corso

I vector

- Come gli array utilizzano *locazioni di memoria contigue*: possiamo accedere ai singoli elementi usando offset rispetto alla posizione base.
- Internamente implementano allocazione dinamica di memoria. Il meccanismo è analogo a quello descritto in precedenza, e per motivi di efficienza solitamente i vector possono allocare un po' di spazio extra per poter accomodare possibili richieste future (notate il trade-off tra spazio e tempo!).
- Se confrontati con gli array, i vector possono consumare un po' di memoria in più (marginale) ma sono molto più efficienti nello svolgere operazioni che coinvolgono la crescita dinamica della sequenza.
- I Vector sono “equipaggiati” con una serie di funzioni di utilizzo (metodi della classe) che li rendono molto comodi da usare

I vector

- per usare i vector dobbiamo includere il corrispondente modulo di libreria

```
#include <vector>
```

- La dichiarazione di una variabile vector prevede la **specificazione del tipo base**

```
vector<T> v;
```

questa dichiarazione indica che v è un vector le cui celle sono di tipo T, dove T deve essere un tipo noto

- In questa dichiarazione il vector costruito è (ancora) vuoto ossia ha 0 celle

Creazione di un vector

- `vector<T> v;` Crea un vettore vuoto, v, privo di elementi
- `vector<T> v(w);` Crea un vettore, v, e inizializza gli elementi in modo che siano uguali a quelli di un vettore w (preesistente) dello stesso tipo (copy constructor).
- `vector<T>v(size);` Crea un vettore v di size elementi di tipo T e li inizializza ad un valore predefinito (che dipende dal tipo)
- `vector<T>v(size, elem);` Crea un vettore v di size elementi di tipo T e li inizializza al valore elem

creazione di un vector - esempio

```
#include <iostream>
#include <vector>

using namespace std;

int main ()
{
    int size;

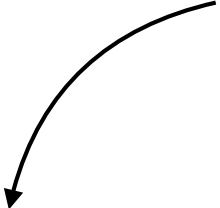
    cout << "Inserisci la dimensione della sequenza " << endl;
    cin >> size;

    vector<int> v1(size);

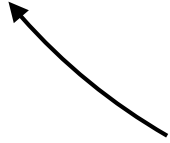
    for (int i=0; i<size; ++i)
        cout << v1[i] << " ";
    cout << endl;

    return(0);
}
```

Come per gli array dinamici la
lunghezza può essere
data a run time



come per gli array `v[index]` restituisce
l'elemento di `v` che si trova nella posizione `index`



Accesso agli elementi di un vector

- la classe `vector` mette a disposizione varie operazioni per la manipolazione dei dati memorizzati (esempi)
 - `v.at(index)` Restituisce l'elemento di `v` che si trova nella posizione `index`. Rispetto all'operatore `[]`, la funzione `at` solleva un'eccezione se `index` è out-of-bound
 - `v.front()` Restituisce il primo elemento (senza verificare se `v` è vuoto, ossia non solleva eccezioni)
 - `v.back()` Restituisce l'ultimo elemento (senza verificare se `v` è vuoto, ossia non solleva eccezioni)

NOTA BENE: `at`, `front`, `back`, ... sono chiamate funzioni membro della classe `vector` (chiamate usando la notazione col punto)

chiamata a funzioni membro

- sintassi

```
nome_variabile.nome_funzione_membro(lista_argomenti)
```

notazione col punto (già vista per
accedere ai campi delle struct)

Capacità di un vector

- `v.size()` Restituisce il numero di elementi di un vector. Corrisponde agli elementi inseriti in un vector, non necessariamente all'effettiva capacità. Il valore restituito è un `unsigned int`
- `v.capacity()` Restituisce l'effettiva dimensione dello spazio di memoria allocato (non necessariamente uguale a `size`)
- `v.max_size()` Restituisce la dimensione massima di un vector che può essere gestita dal sistema/librerie

Capacità di un vector

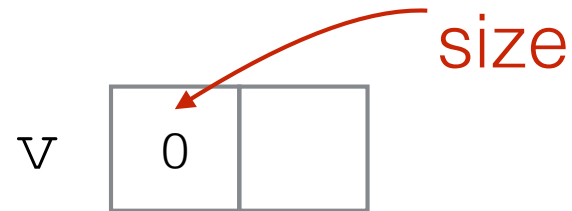
- `v.resize(n)` modifica la dimensione di `v`. Se `v.size < n` richiede spazio in più se necessario, altrimenti elimina gli elementi non necessari
- `v.empty()` verifica se `v` è vuoto (nel caso ritorna `true` altrimenti ritorna `false`)

Modificatori

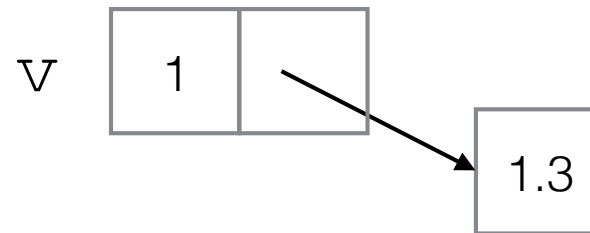
- `v.push_back(elem)` Una copia di `elem` viene inserita in fondo `v` (e la `size` di `v` aumenta di 1, mentre `capacity` aumenta solo se necessario)
- `v.pop_back()` Elimina l'ultimo elemento di `v` (la dimensione di `v` diminuisce di 1)
- `v.clear()` Elimina tutti gli elementi da `v` e lo lascia di dimensione 0

Una rappresentazione grafica semplificata per i vector

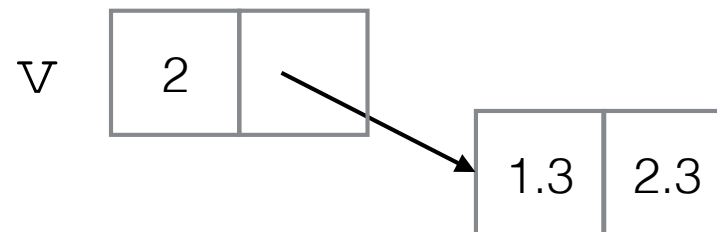
```
vector<double> v;
```



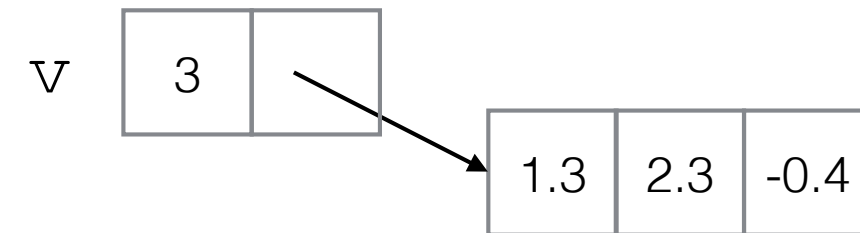
```
v.push_back(1.3);
```



```
v.push_back(2.3);
```



```
v.push_back(-0.4);
```



Esempio: capacità di un vector

```
#include <iostream>
#include <vector>

using namespace std;

int main ()
{
    std::vector<int> v1;

    int mysize;

    cout << "Inserisci le dimensioni del vector: " << endl;
    cin >> mysize;

    for (int i=0; i<mysize; ++i) {
        cout << "size: " << v1.size() << "\t";
        cout << "capacity: " << v1.capacity() << endl;
        v1.push_back(i);
    }

}
```

Esempio: capacità di un vector (OUTPUT)

Inserisci le dimensioni del vector:

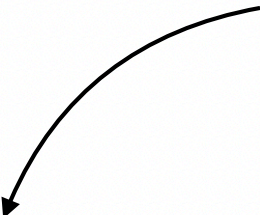
10

size:	0	capacity:	0
size:	1	capacity:	1
size:	2	capacity:	2
size:	3	capacity:	4
size:	4	capacity:	4
size:	5	capacity:	8
size:	6	capacity:	8
size:	7	capacity:	8
size:	8	capacity:	8
size:	9	capacity:	16

Dopo la dichiarazione



Ad ogni push_back capacity
si adatta crescendo alla
potenza di 2 (superiore) più
vicina



vector e funzioni

- una funzione può avere un vector come parametro formale, passato per valore, per riferimento, per riferimento costante
- Un ESEMPIO:

```
void reverse_vector(const vector<int>& v, vector<int>& v_rev) {  
    if (v==v_rev) throw EXC;  
    if (v.empty()) {v_rev.clear(); return;}  
  
    if (v.size() != v_rev.size()) v_rev.resize(v.size());  
    for (int i=0; i<v.size(); ++i)  
        v_rev.at(v_rev.size()-1-i)=v.at(i);  
};
```

vector e funzioni

- una funzione può avere un vector come parametro formale, passato per valore, per riferimento, per riferimento costante

- Meglio ancora:

I VECTOR possono crescere molto,
vogliamo ridurre le copie

```
vector<int> reverse_vector(const vector <int> & v) {  
    vector<int> v_rev(v.size());  
  
    for (int i=0; i<v.size(); ++i)  
        v_rev.at(v_rev.size()-1-i)=v.at(i);  
    return v_rev;  
};
```


vector e memoria (cenni)

- il vector utilizza memoria allocata dinamicamente:

```
vector<int>v(1000);  
cout << sizeof(v) << endl;
```

- La richiesta di spazio heap avviene come sempre attraverso il comando new. La richiesta viene fatta implicitamente da una funzione membro chiamata costruttore invocata all'atto della dichiarazione di una variabile vector.
- La new viene chiamata quando serve anche da altre funzioni (resize, push_back, ...)

come fare a liberare lo spazio quando non è più necessario? Il vector è dotato di una funzione membro chiamata distruttore che viene invocata implicitamente quando un vector finisce out of scope. Usando i vector non corriamo il rischio di provocare *memory leak*.

vector e operatori relazionali

- `vector<T> v1, v2;`
....
- `(v1==v2)` controlla prima le dimensioni e se corrispondono verifica gli elementi
- `(v1<v2)` confronto lessicografico: controlla prima le dimensioni e se corrispondono verifica elemento per elemento fermandosi al primo elemento che non rispetta la disuguaglianza

RICORDATE: questo non era fattibile con gli array dinamici!

Gli operatori sono stati “riprogettati” per poter funzionare sui vector

copie

- Le copie tra vector sono **copie profonde**:
 - costruttore di copia (copio mentre dichiaro il nuovo vector)
`vector<T> V_copy(V);`
 - assignment: `V_copy=V;`
- (State attenti ai vector di puntatori, perche' copiano i puntatori, ma non gli elementi a cui essi puntano)

vector e string

- i vector sono solo uno dei molti tipi di contenitori generici che si possono trovare nelle Standard Libraries
- le string sono “quasi” dei container, nel senso che non sono generici, contengono solo sequenze di caratteri e sono equipaggiati di una grande quantità di operazioni / funzioni specifiche

string

`#include<string>`

- sono dotate di molte operazioni
- `s1=s2` // assegnazione
`s +=x` // append
`s1+s2` // concatenazione
`s1==s2` ma anche `s1 < s2 ...` // confronto
- `s.size()` // numero di caratteri in s
`s.length()` //numero di caratteri in s
`s.c_str()` // l'equivalente c-string
`s.begin()` //primo carattere
-