heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

ResizableArray

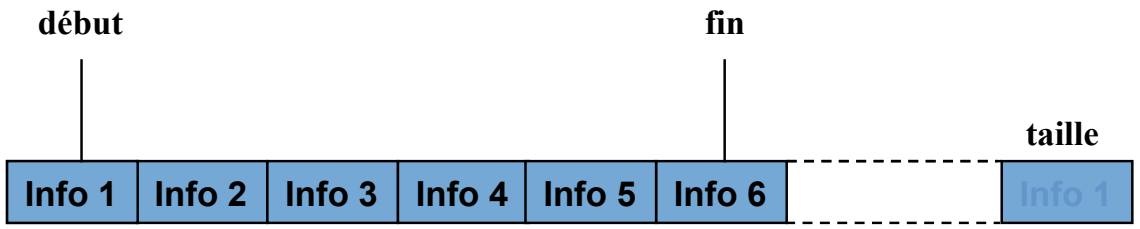
Interface (1)

```
template < typename T >
class ResizableArray
{
public: // types
    using value_type = T;
    using reference = T&;
    using const_reference = const T&;
    using pointer = T*;

private: // attributs
    pointer _begin;
    pointer _end;
    pointer _end_cap;

public: // constructeurs / destructeurs / affectations
    ResizableArray(size_t size = 0);
    ResizableArray(const ResizableArray& other);
    ResizableArray(ResizableArray&& other);
    ResizableArray& operator = (const ResizableArray& other);
    ResizableArray& operator = (ResizableArray&& other);
    ~ResizableArray();
```

Interface (2)

```
    void swap(ResizableArray& other) noexcept;

public: // gestion taille / capacité
    size_t size() const noexcept;
    size_t capacity() const noexcept;
    void resize(size_t newSize);
    void reserve(size_t newCapacity);
    void shrinkToFit();

public: // gestion élément en queue
    void push_back(const_reference value);
    void pop_back();
    reference back();
    const_reference back() const;

public: // gestion élément via son index
    void insert(size_t pos, const_reference value);
    void erase(size_t pos);
    reference at(size_t pos);
    const_reference at(size_t pos) const ;
};
```

Rappel - taille et capacité




```
const int CAPACITY = 10;

struct tableauDeInt {
    int data[CAPACITY];
    size_t size;
};
```

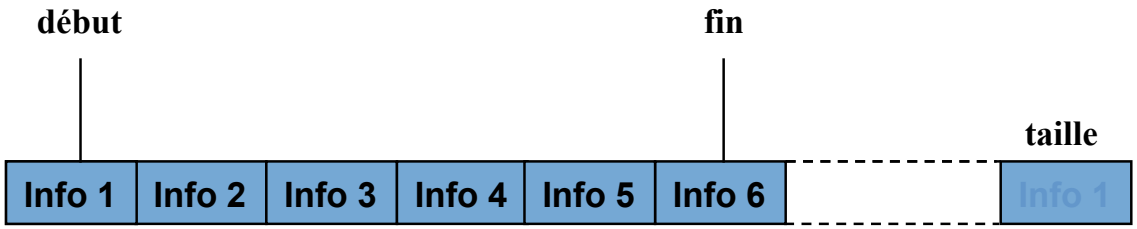
- Voir exemple de mise en oeuvre de pile dans l'introduction
- Consultation et modification du n^{ième} élément en O(1)
- Insertion/suppression en fin en O(1)

```
void push_back(int e) {
    data[size++] = e;
}
```

```
void pop_back() {
    --size;
}
```

- Insertion/suppression en position n en O(size-n) en déplaçant vers la droite/gauche tous les éléments en position n ou plus.
- Capacité fixe - lève une erreur si taille demandée > capacité

Rappel - Tableau de capacité variable




```
struct vecteurDeInt {
    int* data;
    size_t size;
    size_T capacity;
};
```

```
struct vecteurDeInt {
    int* begin;
    int* end;
    int* capacity;
};
```

- Comme pour le tableau de capacité fixe, on alloue (capacité) parfois plus que nécessaire (taille).
- Mais … le tableau est alloué dynamiquement plutôt que statiquement, ce qui permet d'augmenter la capacité si nécessaire.
- La complexité de la gestion dynamique de la capacité s'ajoute à la complexité d'insertion dans un tableau

Rappel - Gestion de la capacité

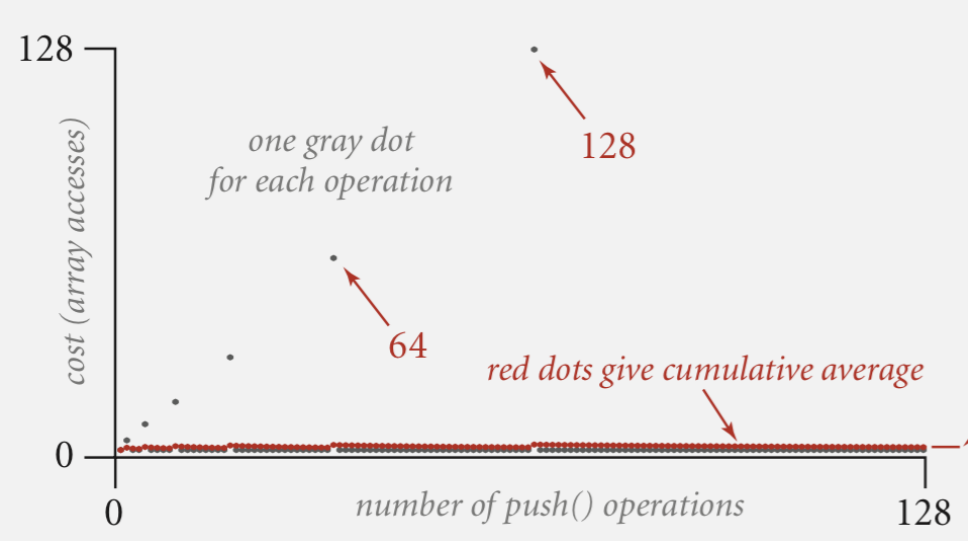
- Coût d'une réallocation + copie d'un tableau de n élément - O(n)

- Stratégie naïve - adapter la capacité à chaque insertion
  - coût pour l'insertion de n éléments dans un tableau vide: $1 + 2 + 3 + \ldots + n = O(n^2)$.

- Stratégie correcte - doubler la capacité à chaque adaptation. Ne pas adapter si pas nécessaire
  - coût pour l'insertion de n éléments dans un tableau vide:

    $1 + 2 + 4 + 8 + \ldots + 2^{(\log_2(n)-1)} = 2n = O(n)$.

```
size_t mCapacity;
size_t mSize;
int* mData;

void incrCapacity(size_t newcap)
{
    mCapacity = newcap;
    int* tmp = new int[newcap];

    for(size_t i=0; i<mSize; ++i)
        tmp[i] = mData[i];

    delete[] mData;
    mData = tmp;
}
```



Gestion de la mémoire

- Comme pour `std::vector`, on sépare les notions de
  - allocation / libération de la mémoire
  - construction / destruction des objets stockés dans le tableau

- La mémoire allouée est liée à la capacité du tableau.
- Le nombre d'objets construits est liée à la taille du tableau

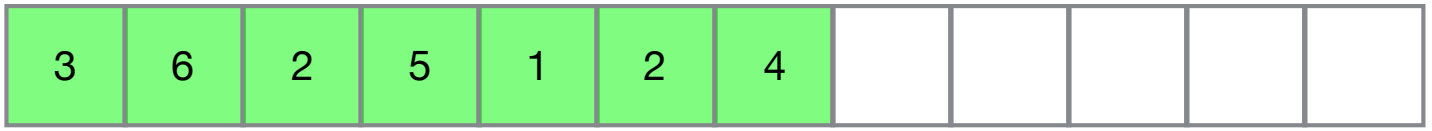
en C++

```
// allocation de mémoire pour N éléments de type TYPE, sans construction

TYPE* ptr = reinterpret_cast<TYPE*>(::operator new (N * sizeof(TYPE)));

// libération de mémoire sans destruction

::operator delete(ptr);

// construction par défaut en place

new(ptr) TYPE();

// construction par copie en place

new(ptr) TYPE(value);

// destruction sans libération de mémoire

ptr->~TYPE();
```

Exemple

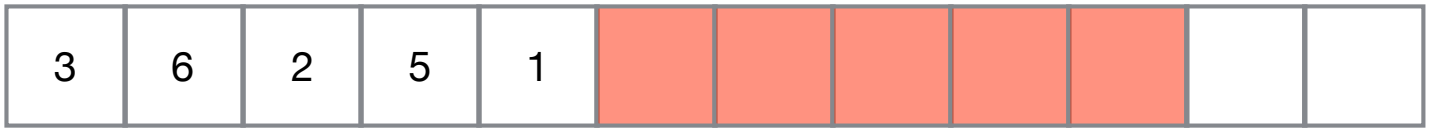
```
ResizableArray<int> T;
…
```


```
T.push_back(4);
```

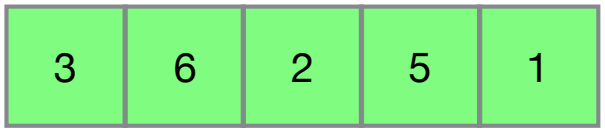



```
T.resize(10);
```

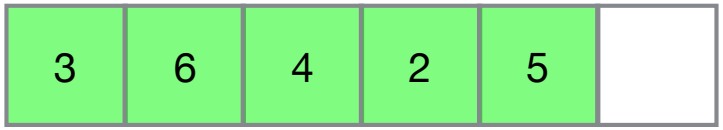

```
T.resize(5);
```



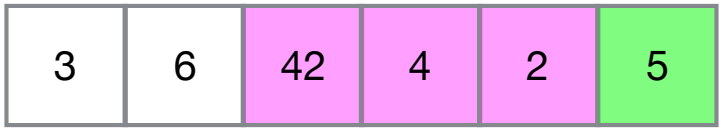

```
T.shrinkToFit();
```



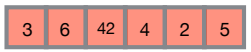

```
T.reserve(6);
```

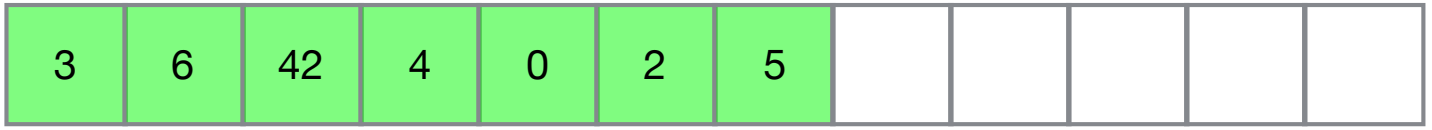



```
T.insert(2,42);
```




```
T.insert(4,0);
```






```
copy constructor
default constructor
operator =
destructor
```

Exceptions

`::operator new`

`copy constructor`

`default constructor`

`operator =`

Peuvent lever des exceptions

Doivent être annulés si une exception a lieu par la suite

`::operator delete`

`destructor`

sont noexcept

Garantie minimale en cas d'exception: pas de fuite de mémoire

- Si plus d'une opération ci-dessus a lieu lors d'une opération sur ResizeArray.
- En cas d'exception lancée par l'une d'entre elles
- Il faut annuler toutes celles qui ont réussi précédemment
- 2 approches possibles: `try/catch/throw` ou RAII