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

```
ResizableArray::ResizableArray(size_t size);
```

# ResizableArray::ResizableArray(size_t size);

- Comment écrire le constructeur ?

- Que doit-il faire?
  - Allouer la mémoire pour **size** éléments de type `T`
  - Initialiser **`_end`** et **`_end_cap`** pour que taille et capacité soient égales à **`size`**
  - Initialiser les **`size`** éléments à leur valeur par défaut
  - Ne pas laisser fuir de mémoire si l'allocation ou l'initialisation des éléments lèvent une exception, et relever l'exception après avoir libérer ses ressources.

# Allocation de mémoire et valeurs de _end et _end_cap

```
ResizableArray(size_t size = 0) {

    _begin = (T*) (::operator new(size * sizeof(T)));
    _end = _begin + size;
    _end_cap = _begin + size;

}
```

- L'allocation se fait avec l'**`::operator new`**. Il fonctionne comme **`malloc`** en C, mais lève l'exception **`std::bad_alloc`** plutôt que de retourner un pointeur nul.
- **`::operator new`** prend le nombre de bytes en paramètre. Il faut donc multiplier **`size`** par **`sizeof(T)`**
- **`_end`** et **`_end_cap`** sont des pointeurs **`T*`**. Il ne faut donc pas multiplier size par **`sizeof(T)`**, l'opérateur **`+`** sur les **`T*`** s'en chargeant.

# Utilisation des alias de type `value_type` et `pointer`

```
ResizableArray(size_t size = 0) {

   _begin = reinterpret_cast<pointer>(
                ::operator new(size * sizeof(value_type)));
   _end = _begin + size;
   _end_cap = _begin + size;

}
```

- Il est plus propre et conforme au style de la STL de ne pas utiliser le type générique `T` mais plutôt les alias de type.
- Pour être précis, le conversion du **`void*`** retourné par **`::operator new`** en **`pointer`** - alias **T*** - est un **reinterpret_cast**, ce que l'on peut noter explicitement.

# Initialisation

```
ResizableArray(size_t size = 0) {

   _begin = reinterpret_cast<pointer>(
                ::operator new(size * sizeof(value_type)));
   _end_cap = _begin + size;


   for (pointer p = _begin; p < _end_cap; ++p) {
      *p = 0;
   }
   _end = _begin + size;

}
```

- Reste à initialiser le contenu du tableau en le parcourant pour mettre chaque élément à la valeur par défaut.
- Par exemple, le code ci-dessus suffit à passer le codecheck 1

# Valeur par défaut

```
ResizableArray(size_t size = 0) {

   _begin = reinterpret_cast<pointer>(
                ::operator new(size * sizeof(value_type)));
   _end_cap = _begin + size;


   for (pointer p = _begin; p < _end_cap; ++p) {
      *p = value_type();
   }
   _end = _begin + size;

}
```

- Rien ne dit que la valeur **`0`** soit la valeur par défaut du type **`T`**, même si c'est le cas de tous les types numériques.
- La valeur par défaut d'une variable de type **`T`** est donnée par **`T()`**

# Affectation vs. Constructeur

```
*p = value_type();
```

- Il n'est pas correct d'initialiser nos éléments comme ci-dessus avec l'opérateur d'affectation
- Pour des éléments de type **`std::string`** par exemple, cela libèrerait la mémoire précédemment utilisée en plus de copier la valeur affectée.
- Le pointeur contenant l'adresse mémoire à libérer n'a pas été initialisé. On libère alors une adresse mémoire aléatoire, ce qui va très probablement crasher le programme.
- Il faut plutôt appeler explicitement le constructeur avec un placement new.

```
new(p) value_type();
```

# Affectation vs. Constructeur

```
ResizableArray(size_t size = 0) {

   _begin = reinterpret_cast<pointer>(
                ::operator new(size * sizeof(value_type)));
   _end_cap = _begin + size;


   for (pointer p = _begin; p < _end_cap; ++p) {
      new(p) value_type();
   }
   _end = _begin + size;

}
```

# Pourquoi utiliser un pointeur p ?

```
ResizableArray(size_t size = 0) {

   _begin = reinterpret_cast<pointer>(
                ::operator new(size * sizeof(value_type)));
   _end_cap = _begin + size;

   for (_end = _begin; _end < _end_cap; ++_end) {
      new(_end) value_type();
   }

}
```

- Notons qu'il n'est pas nécessaire d'utiliser une variable locale p pour parcourir les éléments.
- _end peut être mis à jour en fur et à mesure de l'initialisation des éléments.

# Traiter les exceptions

- Deux lignes sont susceptibles de lever une exception

```
_begin = reinterpret_cast<pointer>(
                ::operator new(size * sizeof(value_type)));
```

```
new(_end) value_type();
```

- La première ne pose pas de problème. Si elle lève, aucune ressource n'a été allouée et on peut laisser l'exception sortir sans plus
- Si la seconde lève une exception, il faudra
  - Détruire les éléments déjà construits par l'exécution partielle de la boucle
  - Libérer la mémoire allouée à l'adresse `_begin`
  - Re-lever l'exception avec `throw;`

# Traiter les exceptions

```
ResizableArray(size_t size = 0) {

   _begin = reinterpret_cast<pointer>(
                ::operator new(size * sizeof(value_type)));
   _end_cap = _begin + size;

   try {
      for (_end = _begin; _end < _end_cap; ++_end) {
         new(_end) value_type();
      }
   } catch(...) {
      for (pointer p = _begin; p < _end; ++p) {
         p->~value_type();
      }
      ::operator delete(_begin);
      throw;
   }
}
```

# Le cas de `size == 0`

```
ResizableArray(size_t size = 0) {

   _begin = size ? reinterpret_cast<pointer>(::operator
                        new(size * sizeof(value_type))) : nullptr;
   _end_cap = _begin + size;

   try {
      for (_end = _begin; _end < _end_cap; ++_end) {
         new(_end) value_type();
      }
   } catch(...) {
      for (pointer p = _begin; p < _end; ++p) {
         p->~value_type();
      }
      ::operator delete(_begin);
      throw;
   }
}
```

- Enfin, pour une taille nulle, il est plus propre de ne pas allouer de mémoire