

Programmation orientée objet

Création et manipulation de
vecteurs

Vecteur

- Un vecteur est une **collection séquentielle d'items du même type**
- On l'utilise comme si c'était un tableau
- *Particularité intéressante:* si on insère un nouvel item dans un vecteur déjà plein, sa **taille sera automatiquement augmentée** afin de pouvoir recevoir ce nouvel item

Vecteur (suite)

- La classe **vector** fait partie de la STL (Standard Template Library) de C++
[cppreference - vector](#)
- Pour utiliser les vecteurs, il faut donc inclure la classe dans le programme:

```
#include <vector>
```

Création d'un vecteur

- La classe **vector** est une classe **générique**, il faut donc toujours spécifier le type des éléments qu'il contiendra lorsqu'on instancie un vecteur:

```
int main() {  
    vector<int> vectInt;  
    vector<Employee> vecEmploye;  
}
```

Taille et capacité

- Il est important de distinguer la taille et la capacité d'un vecteur
- Un vecteur est un *conteneur séquentiel* qui maintient à l'intérieur un **tableau**
- **Taille**: nombre d'éléments effectivement **contenus** dans le tableau interne. Elle est accessible en utilisant la méthode **size()**
- **Capacité**: taille du tableau **interne**. Elle est accessible en utilisant la méthode **capacity()**

Taille \leq capacité

Insertion et retrait d'éléments

- Étant donné qu'un vecteur est basé sur un tableau interne, on le remplit et le vide **par la fin** de manière efficace:
 - **push_back()**: Ajoute un élément à la fin du vecteur
 - **pop_back()**: Retire le dernier élément du vecteur

Insertion et retrait d'éléments (suite)

- Les fonctions **insert()** et **erase()** permettent d'insérer et retirer des éléments n'importe où dans le vecteur
- Il faut **éviter de faire appel à ces méthodes**, à cause de leur coût en temps d'exécution puisqu'elles font le décalage complet des éléments du tableau qui suivent l'emplacement d'insertion ou de retrait
- Si vous devez fréquemment les appeler, c'est que **l'utilisation d'un vecteur n'est pas appropriée**

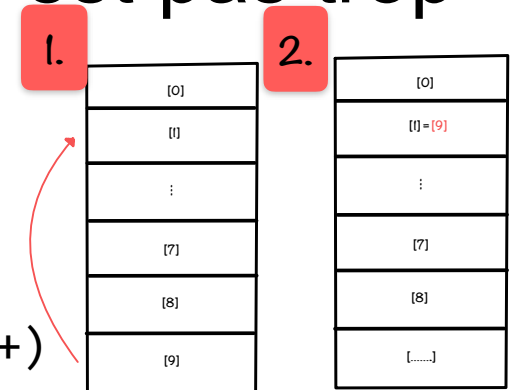
Retrait d'un élément au milieu

- Si l'ordre des éléments n'a pas d'importance, l'opération n'est pas trop coûteuse:

```
int main() {  
    vector<int> vectInt;
```

```
    for (unsigned int i = 0; i < 10; i++)  
        vectInt.push_back(i);
```

```
    vectInt[5] = vectInt.back();  
    vectInt.pop_back();  
}
```



3.

Accès aux éléments

- Étant donné que les éléments sont stockés de manière contigue, on peut accéder de manière efficace aux éléments si on connaît leur index avec:
 - L'opérateur d'accès aléatoire [] comme pour un tableau
 - La méthode **at()**
- Le premier et le dernier élément sont également accessibles en utilisant les méthodes **front()** et **back()**

Manipulation d'un vecteur

```
int main() {  
    vector<int> vectInt;  
    vectInt.push_back(0);  
    vectInt.push_back(1);  
    cout << vectInt.size() << endl; // 2  
    cout << vectInt[0] << endl; // 0  
    cout << vectInt.at(0) << endl; // 0  
    cout << vectInt.front() << endl; // 0  
    cout << vectInt.back() << endl; // 1  
  
    vectInt.pop_back();  
    vectInt.pop_back();  
    cout << vectInt.size() << endl; // 0  
}
```

Ajout d'éléments

Accès aux éléments

Retrait d'éléments

Itération sur les éléments

- On peut itérer sur l'intervalle complet du vecteur en utilisant une boucle for

```
int main() {  
    vector<int> vectInt;  
  
    for(unsigned int i = 0; i < 10; i++)  
        vectInt.push_back(i);  
  
    //Affiche 0 1 2 3 4 5 6 7 8 9  
    for (unsigned int i : vectInt)  
        cout << i << " ";  
  
    cout << endl;  
}
```

Programmation orientée objet

Capacité d'un vecteur

Capacité

- Par défaut, la capacité d'un vecteur est 0
- La capacité d'un vecteur est augmentée automatiquement lorsqu'il n'y a plus assez de place dans le vecteur lorsqu'on tente d'ajouter un nouvel élément
- On peut **initialiser la taille** d'un vecteur lors de sa déclaration (dans ce cas, il sera rempli par des valeurs par défaut ou en appelant le **constructeur par défaut** dans le cas d'un vecteur d'objets)
- Si on sait qu'un indice est inférieur à la capacité d'un vecteur, on peut modifier la valeur à cette position comme dans un tableau (**soyez prudent!**)

Capacité (suite)

```
int main() {  
    vector<int> unTableau(10);  
    vector<Employee> listEmployees(10);  
  
    unTableau[6] = 64;  
    listEmployees[3] = Employee("John", 15000);  
  
    unTableau[10] = 10;  
}
```

Cause une erreur d'exécution puisque l'index dépasse la capacité du vecteur

Augmentation de la capacité

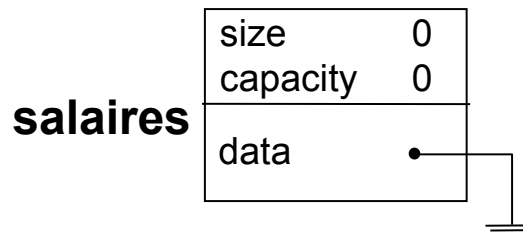
- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```

Augmentation de la capacité (suite)

- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```

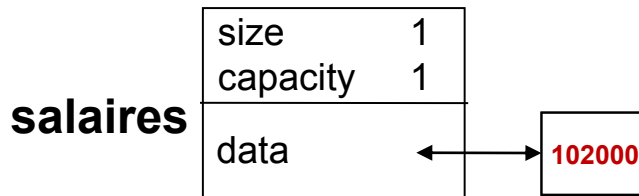


Le vecteur est créé avec capacité nulle.

Augmentation de la capacité (suite)

- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```

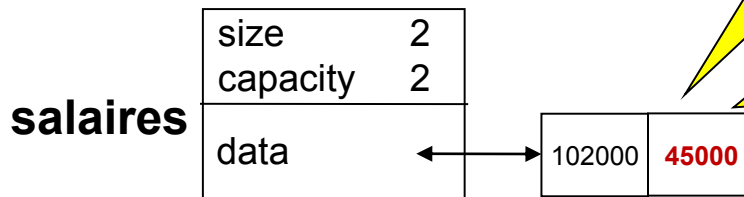


On alloue un tableau de
taille 1 pour contenir le
nouvel élément.

Augmentation de la capacité (suite)

- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```



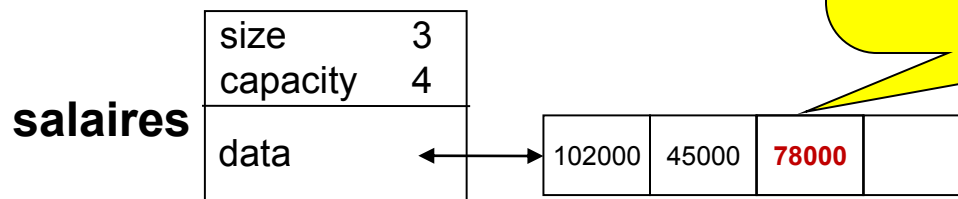
Comme le tableau est plein, on augmente sa taille pour pouvoir ajouter le nouvel élément.

(1) allouer l'espace double sur le tas, (2) copier tout vers cette espace, et (3) détruire l'espace originale

Augmentation de la capacité (suite)

- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```

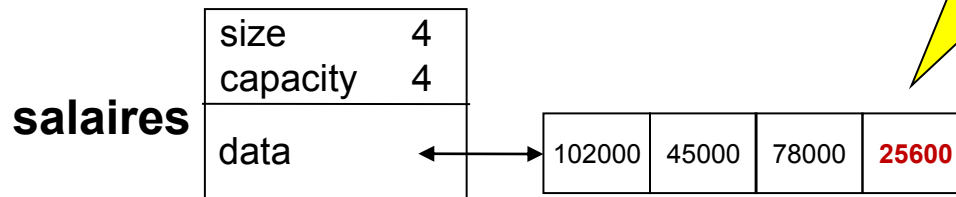


Encore une fois, comme le tableau est plein, on augmente sa taille pour pouvoir ajouter le nouvel élément.

Augmentation de la capacité (suite)

- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```



On ajoute le nouvel élément.

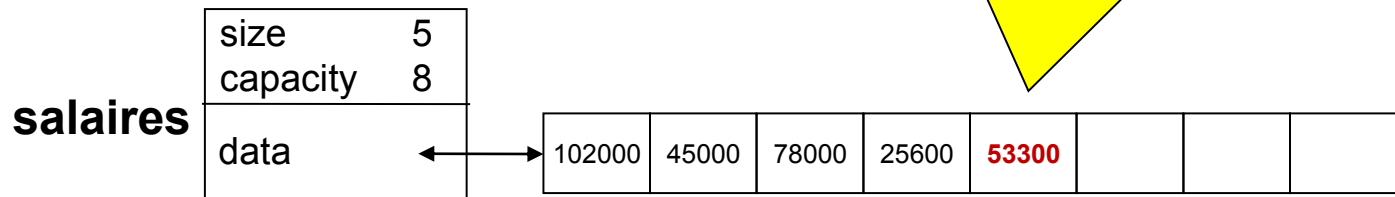
Remarquez qu'on n'a pas atteint la capacité maximale du tableau.

Augmentation de la capacité (suite)

- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```

Encore une fois, comme le tableau est plein, on augmente sa taille pour pouvoir ajouter le nouvel élément.

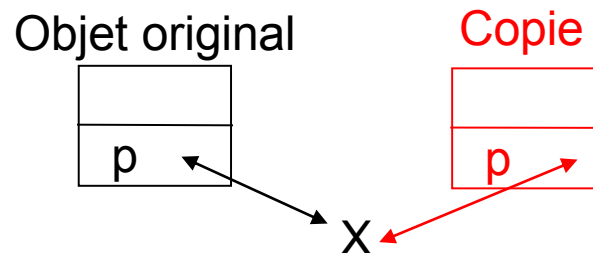


Programmation orientée objet

Copie et référencement de
vecteurs

Shallow copy

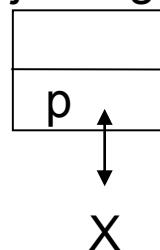
- Jusqu'à maintenant, nous avons toujours considéré que les objets sont copiés de manière superficielle ("**shallow copy**")
- Lorsqu'on fait une copie superficielle, l'objet est copié tel quel. Si un objet contient un pointeur, celui-ci sera partagé entre l'objet original et la copie



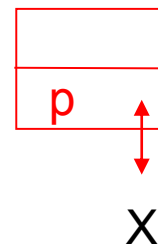
Deep copy

- Une copie en profondeur copie le contenu du pointeur vers un nouvel emplacement mémoire
- Lorsqu'une copie en profondeur est faite, il n'y a aucun partage de mémoire entre l'objet original et la copie

Objet original



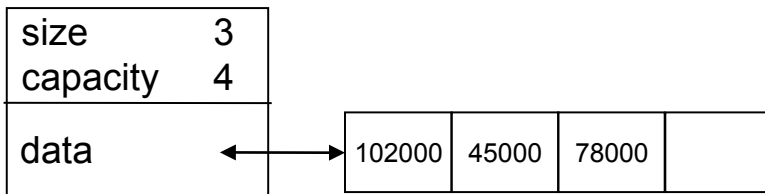
Copie



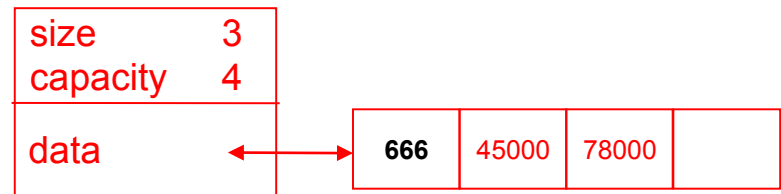
Copie d'un vecteur

- Lorsqu'un vecteur est copié, il s'agit toujours d'une copie en profondeur
- En effet, lorsqu'une copie est réalisée, ce n'est pas le pointeur qu'on copie, mais tout le tableau dynamique
- Ainsi, toute modification à la copie n'a aucun impact sur le vecteur original

Vecteur original



Copie



Passage de vecteur en paramètre

- Lorsqu' on passe un vecteur par valeur à une fonction, une copie éventuellement coûteuse est réalisée
- Il est donc **généralement plus approprié de passer un vecteur par référence**
- Si on veut éviter qu' il soit modifié, on passe une référence constante:

```
void f(const vector<int>& unVecteur)
{
    ...
}
```

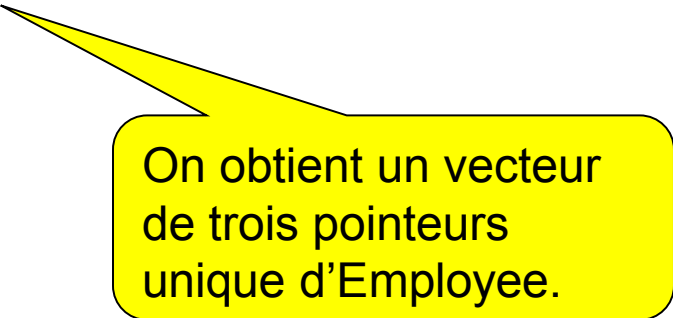
Remarques sur les vecteurs d'objets

- Si on a un vecteur d'objets, l'appel à **push_back()** fera une copie de l'objet dans le vecteur, ce qui peut devenir coûteux
- C'est pourquoi on a souvent tendance, en C++, à **utiliser des vecteurs de pointeurs sur des objets**

Remarques sur les vecteurs d'objets (suite)

- Exemple:

```
int main() {  
    vector<unique_ptr<Employee>> employees;  
    employees.push_back(make_unique<Employee>("John", 15000));  
    employees.push_back(make_unique<Employee>("Mark", 16000));  
    employees.push_back(make_unique<Employee>("Jenny", 12000));  
}
```



On obtient un vecteur de trois pointeurs unique d'Employee.