Programmation orientée objet

Allocation dynamique

Allocation automatique et allocation dynamique

- Automatique:

```
Employee e1("John", 15000);
Employee e1 = Employee ("John", 15000);
```

- Dynamique:

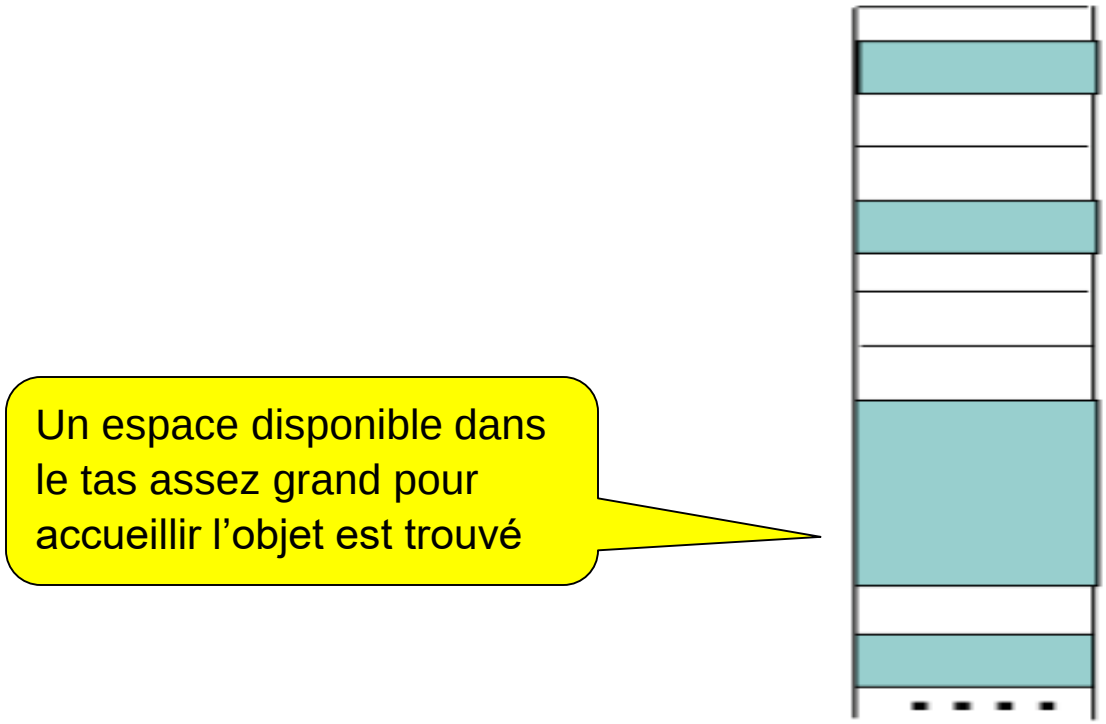
```
Employee * e1;
e1 = new Employee ("John", 15000);
```

Étapes de l'allocation d'un pointeur

```
Employee* e1 = new Employee ("John", 15000);
```

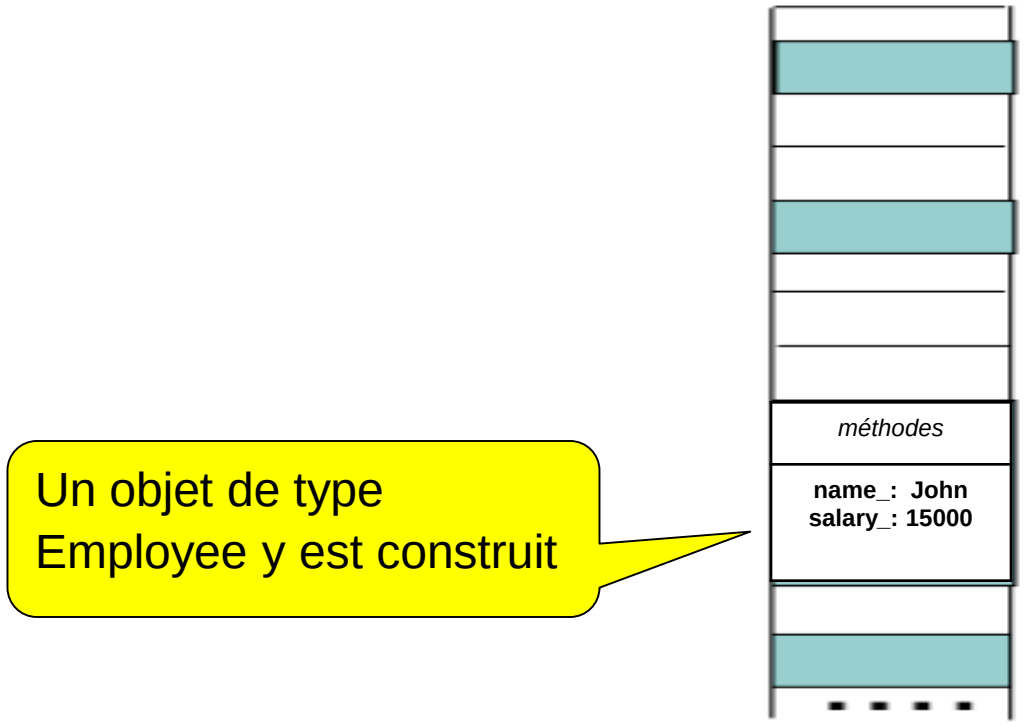
Étapes de l'allocation d'un pointeur

```
Employee* e1 = new Employee ("John", 15000);
```



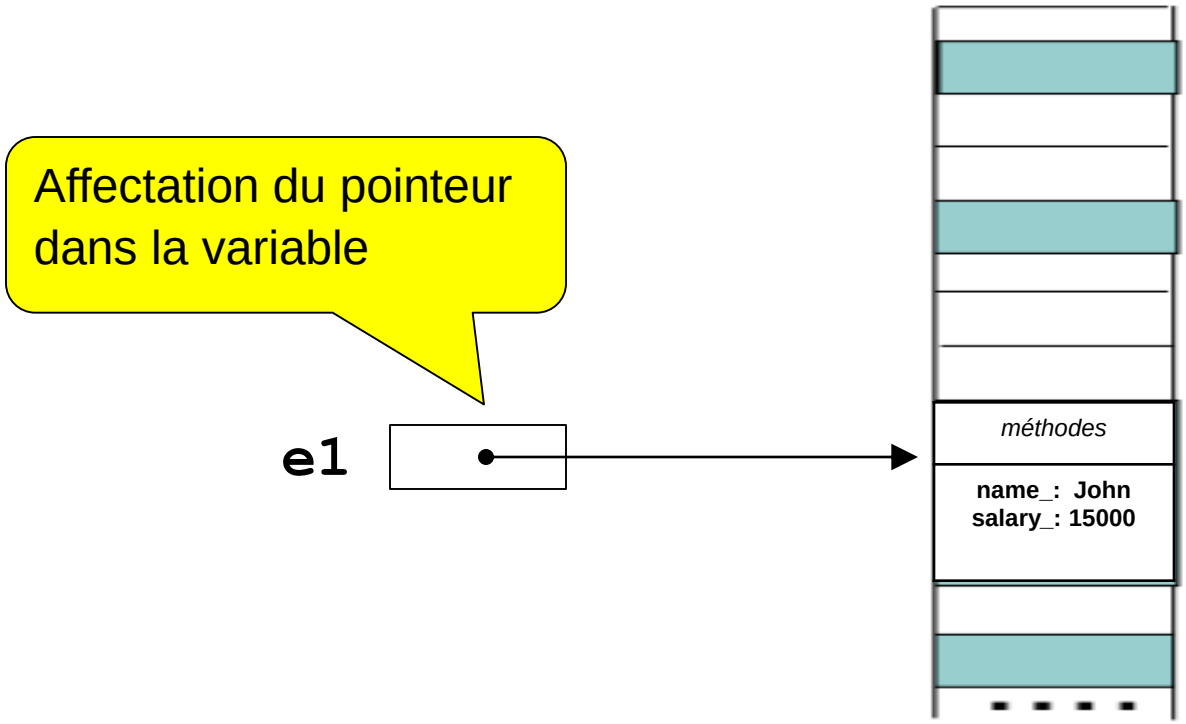
Étapes de l'allocation d'un pointeur

```
Employee* e1 = new Employee ("John", 15000);
```



Étapes de l'allocation d'un pointeur

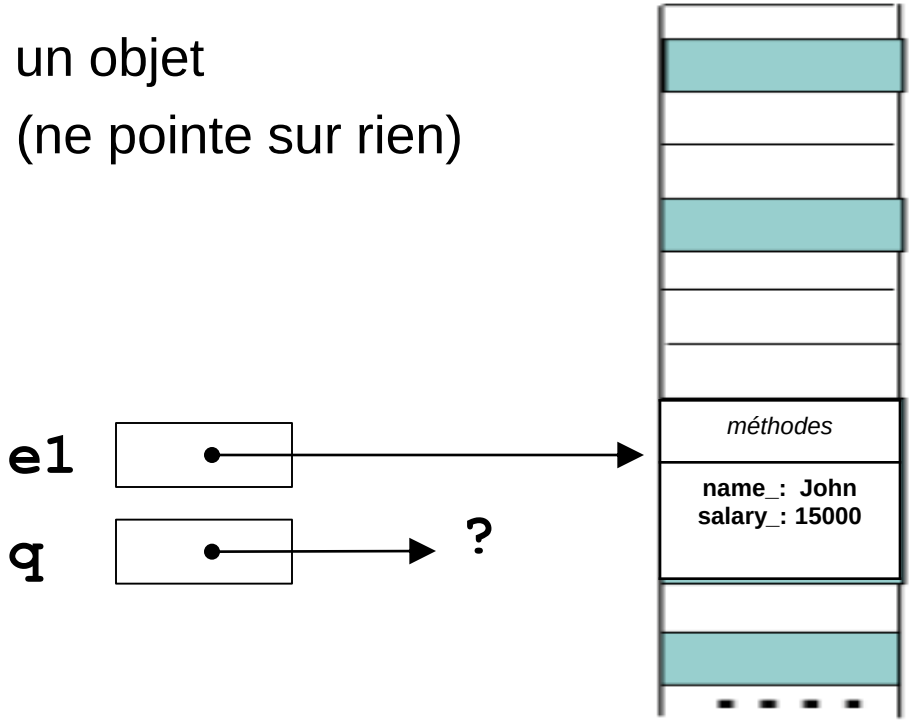
```
Employee* e1 = new Employee ("John", 15000);
```



Pointeurs

```
Employee* e1 = new Employee ("John", 15000);
Employee* q;
```

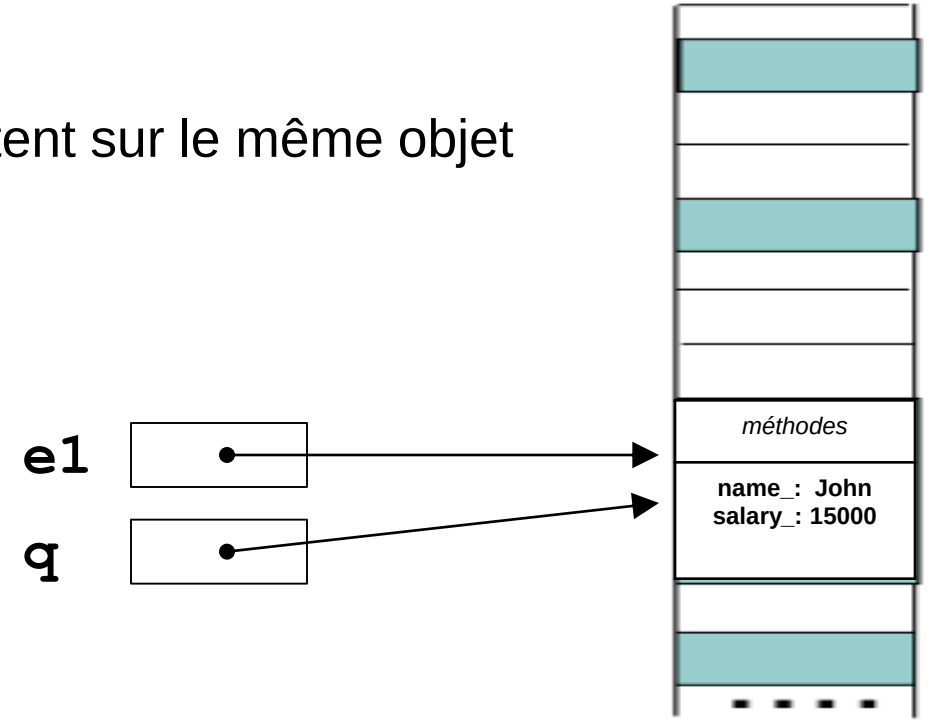
- Le pointeur **e1** pointe sur un objet
- Le pointeur **q** est invalide (ne pointe sur rien)



Pointeurs

```
Employee* e1 = new Employee ("John", 15000);
Employee* q;
q = e1;
```

- Les deux pointeurs pointent sur le même objet



Initialisation des pointeurs

- Prenez l'habitude de toujours initialiser vos pointeurs:

```
Employee* e1 = nullptr; //C++11
```

Initialisation des pointeurs

- Dans un constructeur aussi, si la classe définit un attribut dynamique:

```
class UneClasse
{
public:
  UneClasse();
...
private:
  Point* monAttribut_;
  ...
};
```

```
UneClasse::UneClasse()
{
  monAttribut_ = nullptr;
}
```

Dé-référencement d'un pointeur

- Si **e1** est un pointeur, l'expression ***e1** retourne l'objet pointé par **e1**
- Si on veut appliquer une méthode de l'objet en question:

```
(*e1).getSalary()
```

- Une autre forme synonyme et plus pratique:

```
e1->getSalary()
```

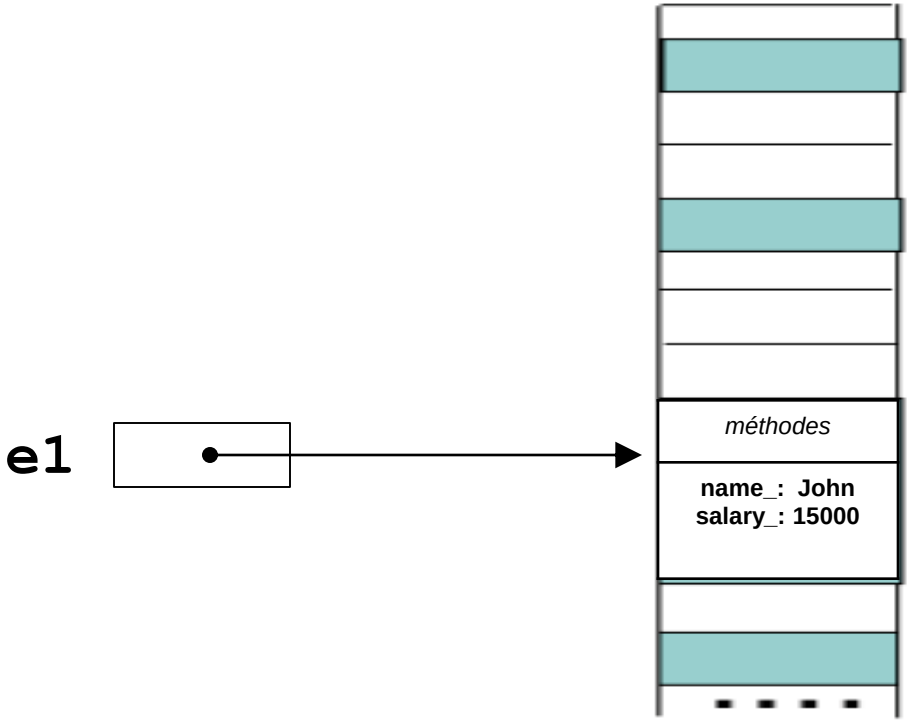
Désallocation de mémoire

- Pour tout appel à **new** il faut retrouver quelque part un appel à **delete** qui désalloue la mémoire:

```
int main()
{
  Employee* e1 = new Employee
  ("John", 15000);
   ...
  delete e1;
}
```

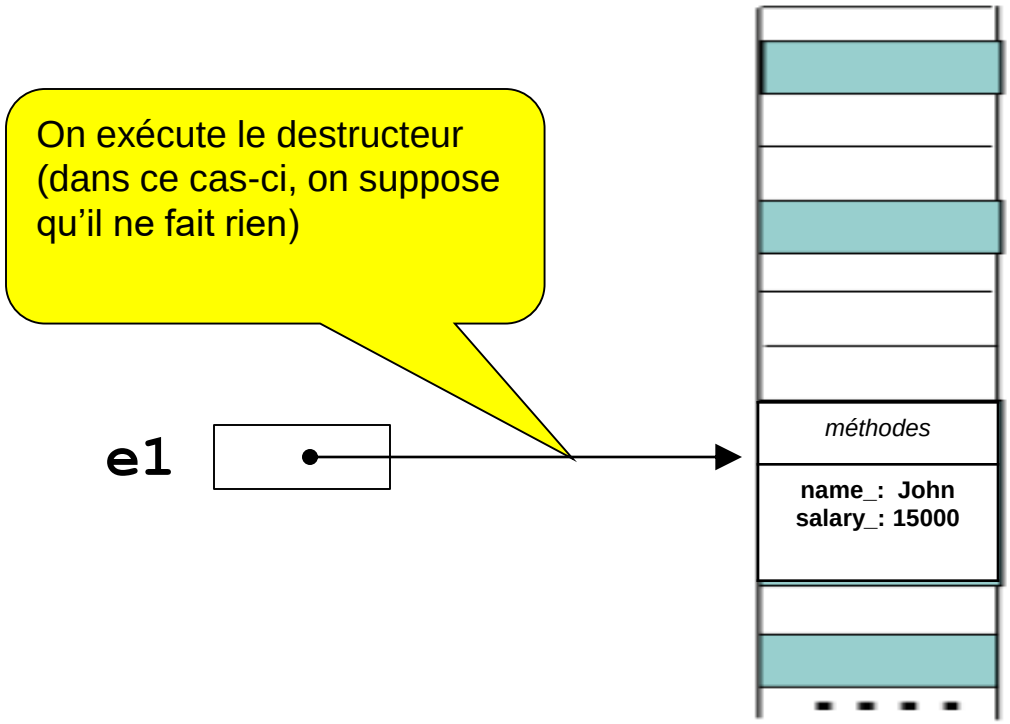
Étapes de la désallocation

```
delete e1;
```



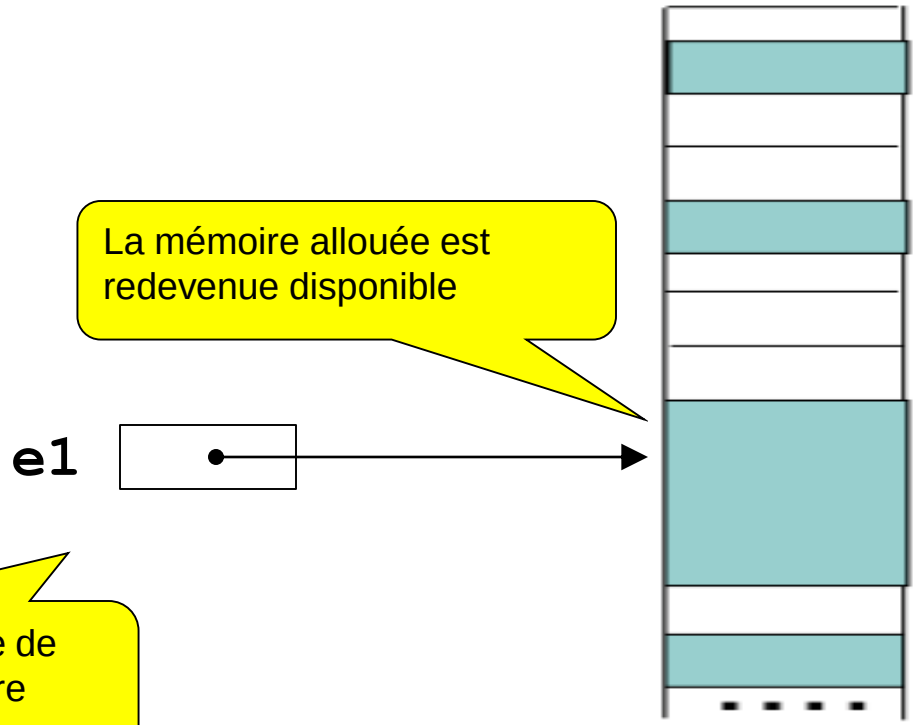
Étapes de la désallocation

```
delete e1;
```



Étapes de la désallocation

```
delete e1;
```



Attention, le pointeur continue de pointer sur un espace mémoire invalide

Désallocation

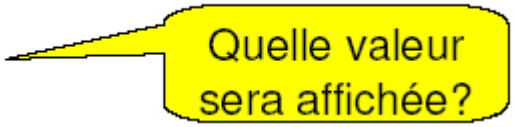
- Ainsi, après l'exécution de **delete**, le pointeur continue de pointer sur le même espace mémoire, devenu invalide
- Pour éviter toute tentative de déréférencer à nouveau ce pointeur, il est bon de le réinitialiser à **nullptr**:

```
delete e1;
e1 = nullptr;
```

Tableaux et pointeurs

- En C++, l'adresse d'un tableau est en fait un pointeur qui pointe sur le premier élément du tableau:

```
int a[5] = {1,2,3,4,5};
int* p = a;
*p = 10;
cout << a[0];
```



Arithmétique des pointeurs

- Si **p** est un pointeur sur un entier, l'expression **p+3** pointe sur le troisième entier en mémoire situé après celui pointé par **p**
- Autre exemple:

```
int a[5] = {1,2,3,4,5};
int* p = a;
++p;
++p;
cout << *p;
```

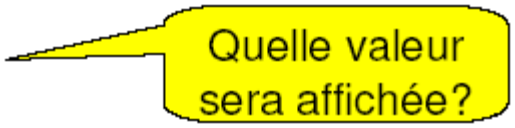


Tableau dynamique

- Considérons l'instruction suivante:

```
int n;
cin >> n;
Employee* listEmployees = new Employee[n];
```

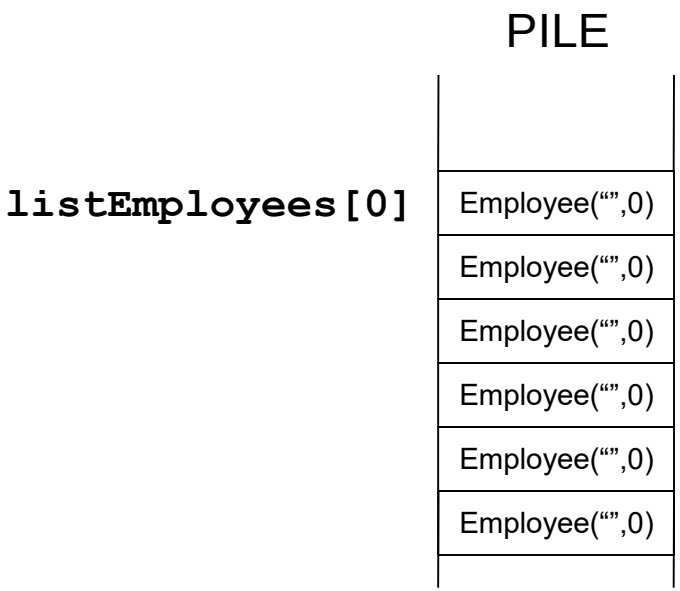
- L'effet de cette instruction est la création sur le tas d'un tableau dynamique dont la taille sera déterminée lors de l'exécution du programme
- Il ne faudra pas oublier de désallouer mémoire:

```
delete[] listEmployees;
```

Récapitulons

- Tableau d'employés alloué automatiquement:

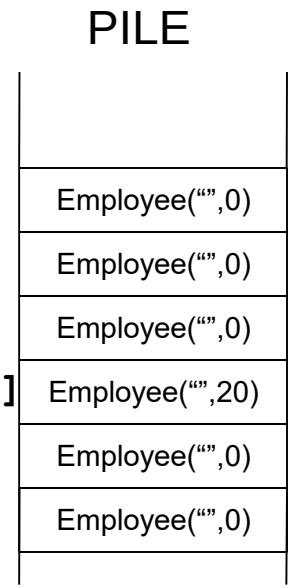
```
Employee
   listEmployees[6];
...
```



Récapitulons

- Tableau d'employés alloué automatiquement:

```
Employee listEmployees[6];
...
listEmployees[3].setSalary(20);
```

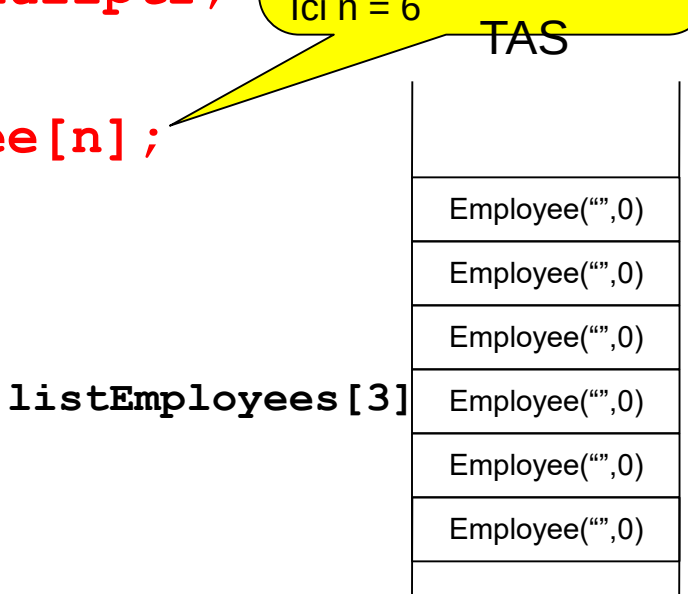


Récapitulons

- Tableau d'employés alloué dynamiquement:

```
int n;
cin >> n;
Employee* listEmployees = nullptr;
...
listEmployees = new Employee[n];
```

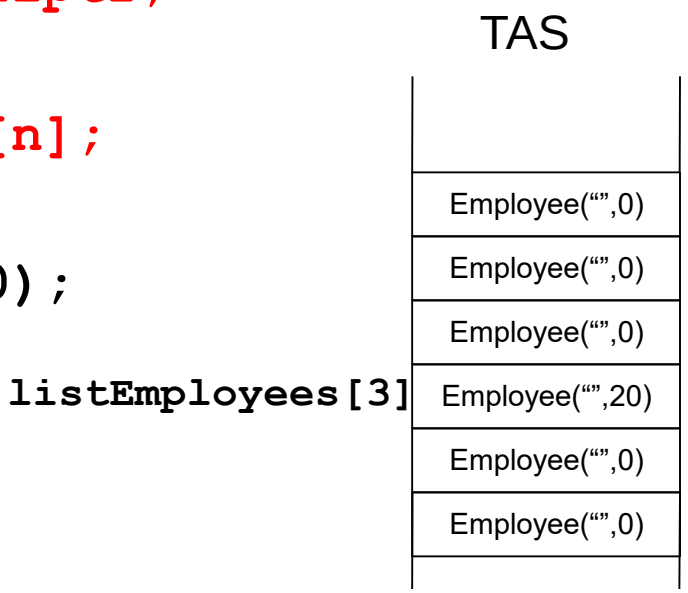
Une séquence de n objets de la classe Employee est ajoutée dans le heap (tas). Ici n = 6



Récapitulons

Tableau d'employés alloué dynamiquement:

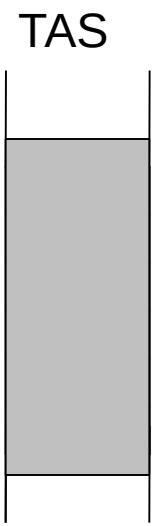
```
Employee* listEmployees = nullptr;
...
listEmployees = new Employee[n];
...
listEmployees[3].setSalary(20);
```



Récapitulons

Tableau d'employés alloué dynamiquement:

```
Employee* listEmployees = nullptr;
...
listEmployees = new Employee[n];
...
listEmployees[3].setSalary(20);
...
delete[] listEmployees ;
listEmployees = nullptr;
```



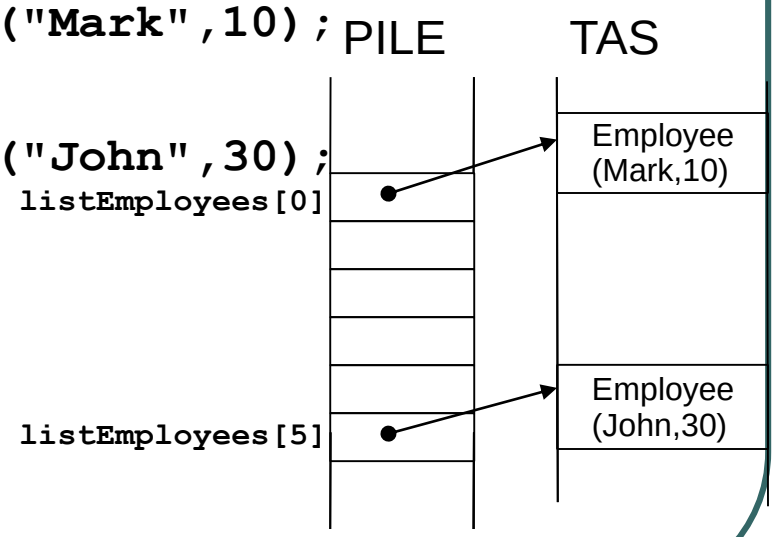
Récapitulons

Tableau de pointeurs d'employés alloué automatiquement:

```
Employee* listEmployees[6];
for ( int i =0 ; i <  6 ; i++)
    listEmployees[i] = nullptr;

listEmployees[0] = new Employee ("Mark",10);
...
listEmployees[5] = new Employee ("John",30);
...
listEmployees[5]->setSalary(20);
...
for (int i = 0; i < 6; ++i) {
    delete listEmployees[i];
}
```

On crée un tableau automatique de 6 pointeurs (qui ne sont pas initialisés).



Récapitulons

Tableau de pointeurs d'employés alloué dynamiquement:

```
Employee** listEmployees;
int n;
cin >> n;
...
listEmployees = new Employee*[n];
for ( int i =0 ; i <  6 ; i++)
    listEmployees[i] = nullptr
.....
listEmployees[0] = new Employee("Mark",10);
...
listEmployees[5] = new Employee(”John”,30);
...
listEmployees[3]->setSalary(20);
...
for (int i = 0; i < 6; ++i) {
    delete listEmployees[i];
}
delete[] listEmployees;
listEmployees = nullptr;
```

Une séquence de n **pointeurs** est ajoutée dans le heap.
Ici n = 6