

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);
```

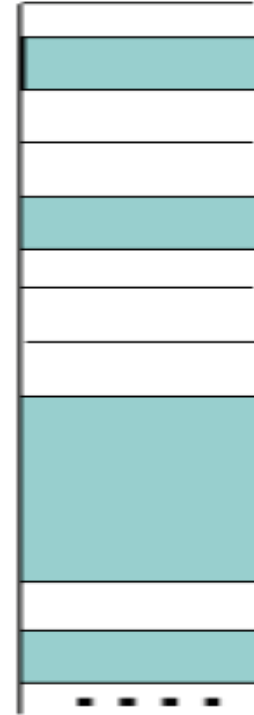
Étapes de l'allocation d'un pointeur

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

Étapes de l'allocation d'un pointeur

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

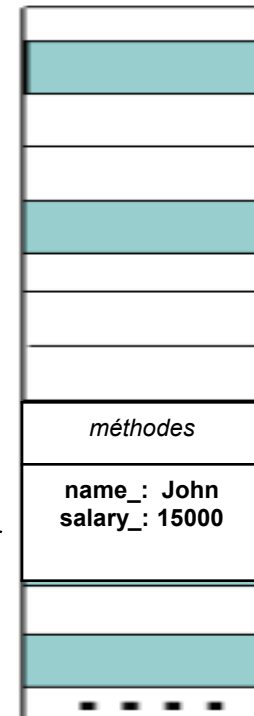
Un espace disponible dans le tas assez grand pour accueillir l'objet est trouvé



Étapes de l'allocation d'un pointeur

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

Un objet de type
Employee y est construit

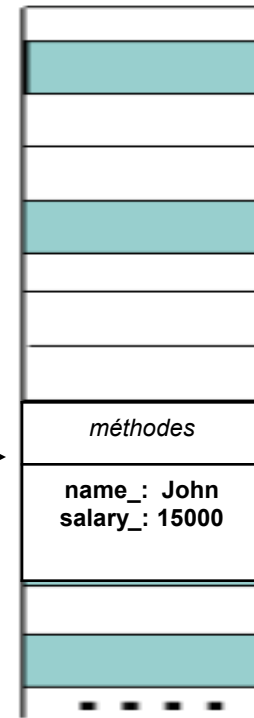
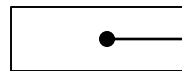


Étapes de l'allocation d'un pointeur

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

Affectation du pointeur
dans la variable

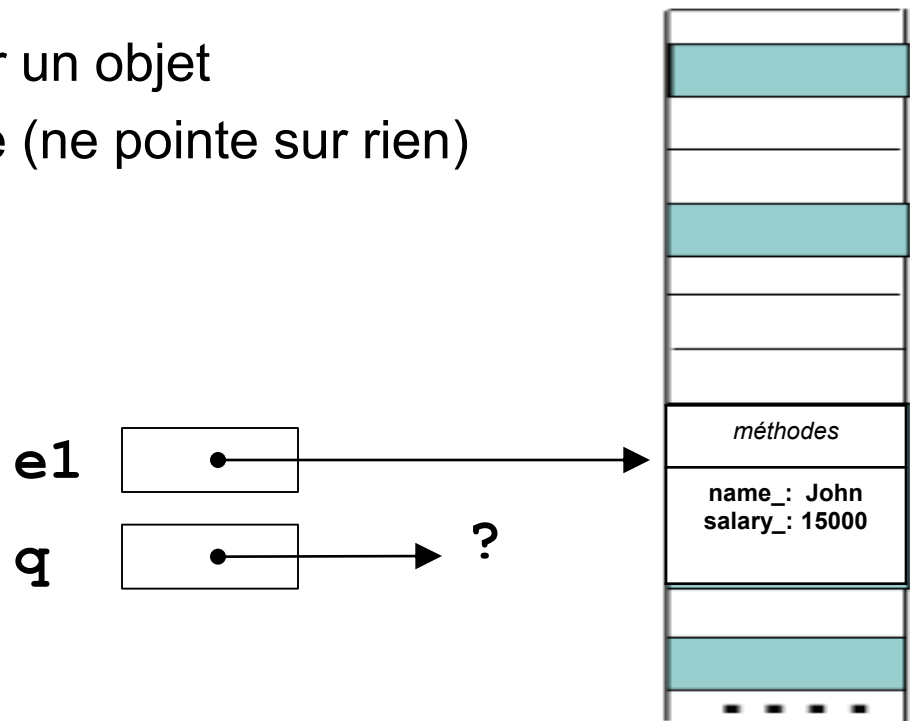
e1



Pointeurs

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

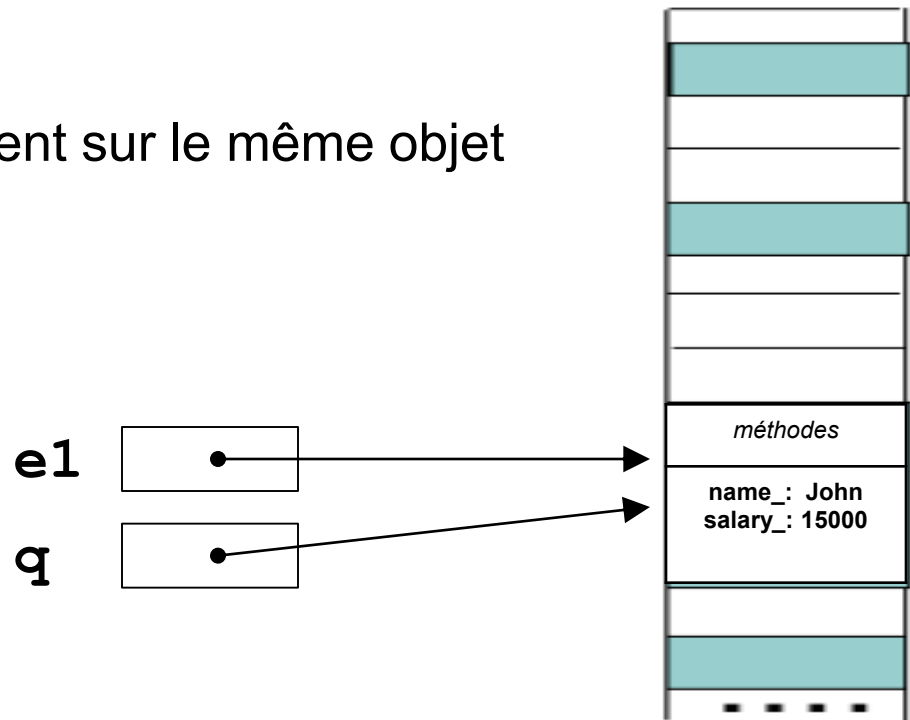
- Le pointeur **e1** pointe sur un objet
- Le pointeur **q** est invalide (ne pointe sur rien)



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()

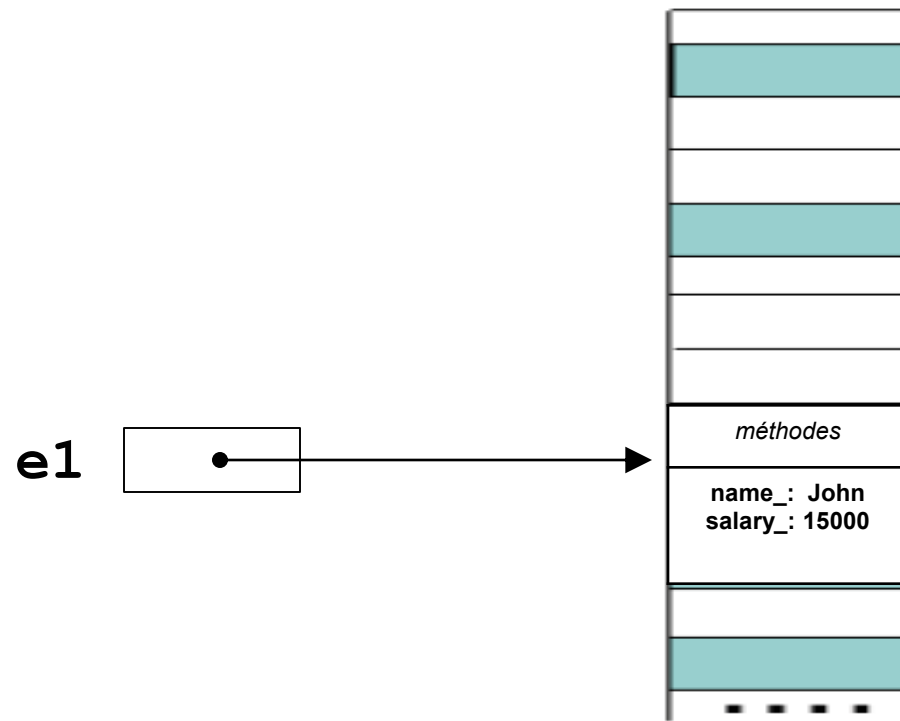
Désallocation de mémoire

- 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;`

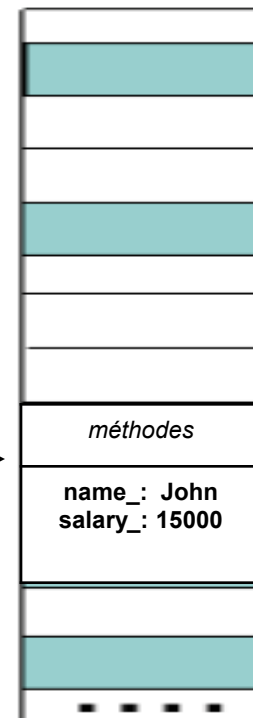
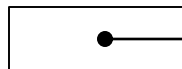


Étapes de la désallocation

`delete e1;`

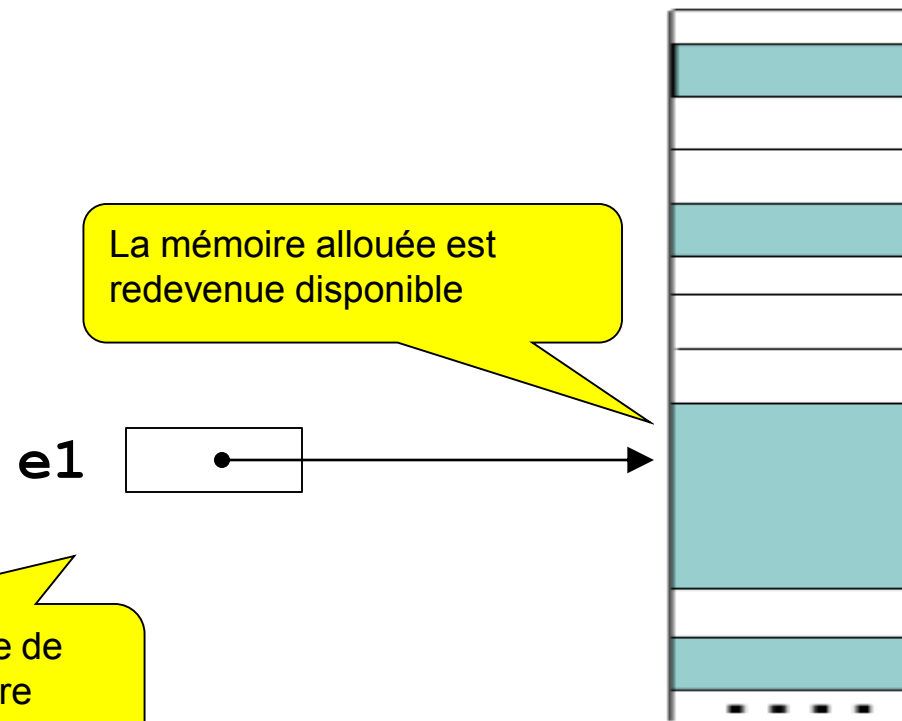
On exécute le destructeur
(dans ce cas-ci, on suppose
qu'il ne fait rien)

`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

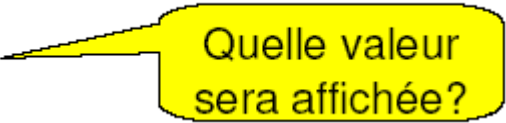
- En C++, l'adresse d'un tableau est en fait un pointeur qui pointe sur le premier élément du tableau:

```
int a[5] = {1,2,3,4,5};
```

```
int* p = a;
```

```
*p = 10;
```

```
cout << a[0];
```

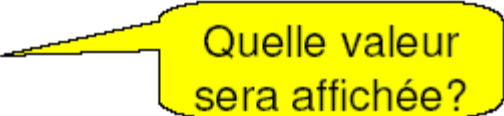
A yellow speech bubble with a black outline, pointing towards the code line `cout << a[0];`. It contains the text "Quelle valeur sera affichée?".

Quelle valeur
sera affichée?

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;
```



Quelle valeur
sera affichée?

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];

...

listEmployees[0]

PILE

Employee("",0)
Employee("",0)
Employee("",0)
Employee("",0)
Employee("",0)
Employee("",0)

Récapitulons

- Tableau d'employés alloué automatiquement:

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

```
...
```

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

```
listEmployees[3]
```

PILE

Employee("",0)
Employee("",0)
Employee("",0)
Employee("",20)
Employee("",0)
Employee("",0)

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

TAS

listEmployees[3]

Employee("",0)

Employee("",0)

Employee("",0)

Employee("",0)

Employee("",0)

Employee("",0)

Récapitulons

Tableau d'employés alloué dynamiquement:

```
Employee* listEmployees = nullptr;
```

```
...
```

```
listEmployees = new Employee[n];
```

```
...
```

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

```
listEmployees[3]
```

TAS

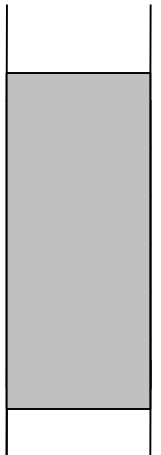
Employee("",0)
Employee("",0)
Employee("",0)
Employee("",20)
Employee("",0)
Employee("",0)

Récapitulons

Tableau d'employés alloué dynamiquement:

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

TAS



Récapitulons

Tableau de pointeurs d'employés alloué automatiquement:

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

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

```
listEmployees[0] = new Employee ("Mark",10);
```

...

```
listEmployees[5] = new Employee ("John",30);
```

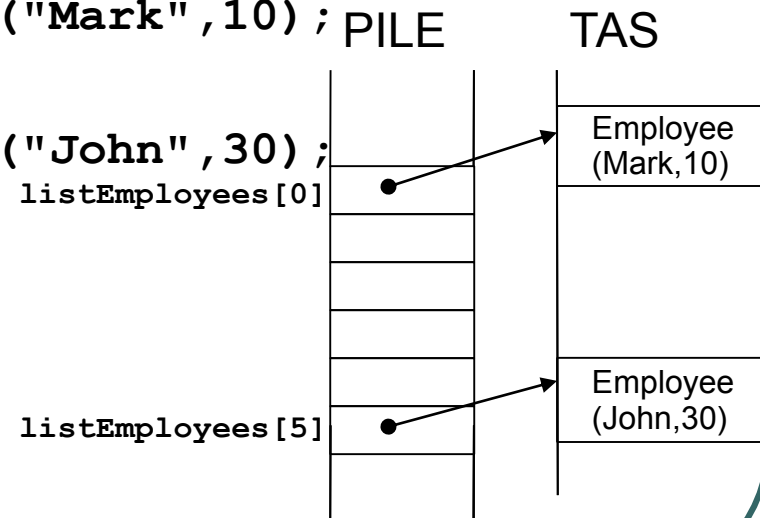
...

```
listEmployees[5]->setSalary(20);
```

...

```
for (int i = 0; i < 6; ++i) {  
    delete listEmployees[i];
```

```
}
```



Récapitulons

Tableau de pointeurs d'employés alloué dynamiquement:

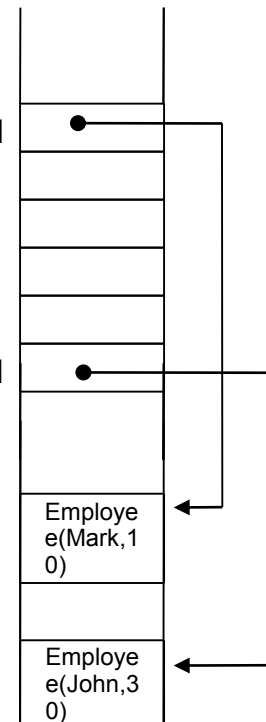
```
Employee** listEmployees;  
int n;  
cin >> n;  
...  
listEmployees = new Employee*[n];  
for ( int i =0 ; i < 6 ; i++)  
    listEmployees[i] = nullptr  
.....  
listEmployees[0] = new Employee("Mark",10);  
...  
listEmployees[5] = new Employee("John",30);  
...  
listEmployees[3]->setSalary(20);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

listEmployees[0]

listEmployees[5]

TAS



Programmation orientée objet

Rappel sur les symboles * et &

Déclaration de variables ou d'objets * et &

- Définition d'un pointeur:

```
int * Ptr = 0;
```

Lors de la définition d'un pointeur, on peut uniquement initialiser la valeur de ce pointeur (et non la valeur vers laquelle il pointe).

Par exemple:

```
float* ptr= nullptr valide
```

```
float* ptr= 10.0 non valide
```

Déclaration de variables ou d'objets &

- Définition d'une référence: permet de manipuler une variable sous un autre nom:

```
int a ;
```

```
int & Ptr = a;
```

Une référence ne peut être ni nulle ni indéterminée, elle doit donc absolument être initialisée lors de la déclaration et la valeur de la référence ne peut pas changer au cours de l'exécution

Section des instructions:

opérateur *

Multiplication:

```
res = a * b;
```

Déréférencement d'un pointeur:

```
*Ptr = 10;
```

Section des instructions:

opérateur &

- Opération Et binaire:

```
res = a & b;
```

- Adresse d'une variable:

Si on a les déclarations suivantes

```
int a ;
```

```
int * Ptr;
```

```
Ptr = &a;
```

Déclaration

Lorsqu'on déclare plusieurs pointeurs/références du même type, il faut répéter le */&.

Par exemple:

`int* a, * b, * c;` (déclare trois pointeurs)

`int* a, b, c;` (déclare un pointeur et 2 entiers)

`int* u, & v = b , w;` (déclare un pointeur, une référence et un entier)