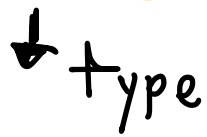
Programmation orientée objet

Classes et objets

Samuel Kadoury

Objet

- Un **objet** est une entité pouvant être créée, stockée, manipulée et détruite
- Un objet possède des attributs (propriétés)
- Un objet offre un certain nombre de méthodes (fonctions membres) pour le manipuler
- Tout objet appartient à une **classe** qui représente un type d'objet



Définition d'une classe en C++

```
class NomDeLaClasse{
public:
    déclarations de constructeurs
    déclarations de méthodes publiques
private:
    déclarations de méthodes privées
    attributs
};
```

→ visibilité

↳ interface.

↳ représentant l'état

Dans une classe

Les méthodes (ou fonctions membres)

La visibilité des membres

L'interface

Exemple de définition de classe

```
class Employee
{
public:
   Employee();
   Employee(string name, double salary);
   double getSalary() const;
   string getName() const;
   void setSalary(double salary);

private:
   string name_;
   double salary_;
};
```

Interface

attributs

État

Remarque: par convention, nous utiliserons toujours un _ pour distinguer les variables qui correspondent aux attributs d'une classe.

Relation Objet - Classe

- La relation qui lie l'objet et sa classe est la même qu'entre une variable et son type :

```
int nbInvités;
```

Variable nbInvités de type int

```
string nom;
```

Objet nom de la classe standard string

```
Employee unEmploye;
```

Objet → on manipule les objets avec les méthodes

Objet unEmploye de la classe Employee

Programmation orientée objet

Méthodes et manipulation d'objets

Samuel Kadoury

Méthodes d'une classe

```
class Employee
{
public:
   Employee();
   Employee(string name, double salary);
   double getSalary() const;
   string getName() const;
   void setSalary(double salary);
   void setName(string name);

private:
   string name_;
   double salary_;
};
```

dans l'interface

Constructeurs

Fonctions d'accès (getters)

Fonctions de modification (setters)

Implémentation des méthodes

- En général, en C++, les méthodes sont implémentées séparément de la définition de classe
- La definition de la classe est faite dans un fichier .h et l'implementation est faite dans un fichier .cpp

On a besoin d'utiliser la signature

Exemple d'implémentation d'une méthode

Dans le .cpp

```
void Employee::setSalary(double salary)
{
  if (salary > salary_)
     salary_ = salary;
}
double Employee::getSalary()
{
   return salary_;
}
```

Paramètres d'une méthode

- Toute méthode a un **paramètre implicite**, qui correspond à l'objet sur lequel elle est appliquée
- Les autres paramètres qui apparaissent dans l'en-tête de la méthode sont les **paramètres explicites**.

Paramètres d'une méthode

- Quand on écrit:
  - `marcel.setSalary(55000);`

    ↳ de la classe employé
- le compilateur crée en fait une fonction à deux paramètres:
  - `setSalary(marcel, 55000);`

Paramètre implicite

Paramètre explicite

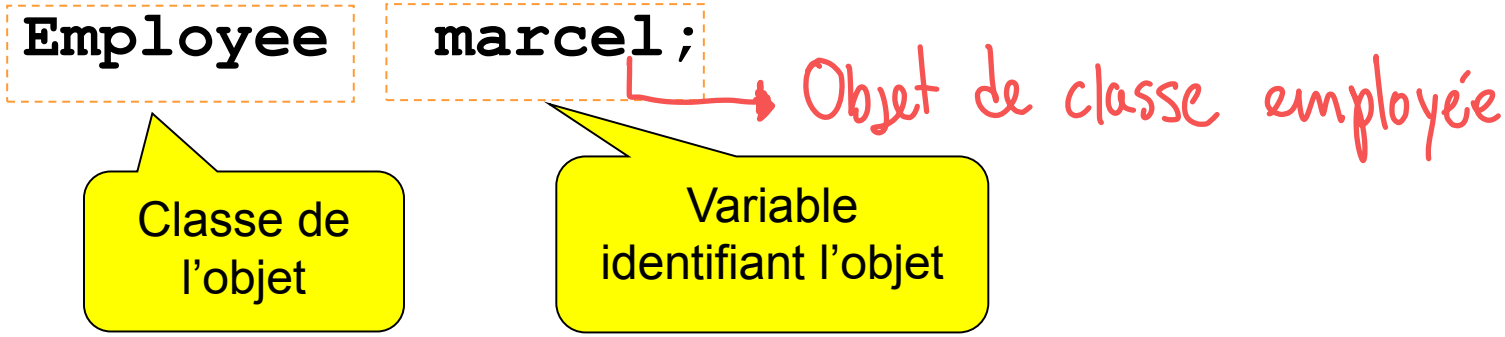
- Mais tout cela est **transparent pour nous**

Principe d'encapsulation

- On n'a pas accès directement aux attributs d'un objet
- On modifie ou on obtient la valeur d'un attribut toujours par l'intermédiaire d'une méthode
- En résumé, on ne peut manipuler l'état d'un objet que par le biais des méthodes qui sont définies par l'interface de sa classe

Manipulation d'un objet

- Création d'un objet:



- Obtention de l'état d'un objet:

```
int salaireMarcel = marcel.getSalary();
```

- Modification de l'état d'un objet:

```
marcel.setSalary(10000);
```

Programmation orientée objet

Constructeurs et destructeurs

Samuel Kadoury

Constructeur

- Le rôle principal d'un constructeur:
  - Initialiser les attributs lors de la création d'un objet
- En général, on a un **constructeur par défaut**, qui ne reçoit aucun paramètre, et qui donne aux attributs des valeurs par défaut
- On peut aussi avoir des **constructeurs par paramètres** qui acceptent comme paramètres les valeurs initiales que l'on veut donner aux attributs

Définition de la classe Employee

```
class Employee
{
public:
   Employee();
   Employee(string name, double salary);
   double getSalary() const;
   string getName() const;
   void setSalary(double salary);
   void setName(string name);

private:
   string name_;
   double salary_;
};
```

Constructeur par défaut

Constructeur par paramètres

Constructeur par défaut

- Exemple d'implémentation du constructeur:

Implementation:

```
Employee::Employee(){
    name_ = "unknown";
    salary_ = 0.0;
}
```

Quand un constructeur par défaut est-il appelé? (suite)

- Appel explicite lors de la simple déclaration d'une variable d'objet:

```
Employee marcel = Employee();
```

```
Employee marcel;
```

Construit un objet de type Employee en fournissant des valeurs par défaut à ses attributs

Forme abrégée de l'initialisation par défaut

Il ne faut **pas** mettre de parenthèses pour l'initialisation par défaut

Quand un constructeur par défaut est-il appelé? (suite)

- Appel implicite lorsqu'on utilise un tableau d'objets:

```
Employee tabDeEmployee[10];
```

Quand un constructeur par défaut est-il appelé? (suite)

- Lorsque l'objet est lui-même un attribut d'un autre objet:

```
class Company
{
public:
    Company ();
   ...
private:
    Employee president_;
    ...
    int nbEmployees_;
};
```

```
Company::Company()
{
    nbEmployees_= 0.0;
}
```

Constructeur par paramètres

- Exemple d'implémentation du constructeur:

```
Employee::Employee(string name, double salary)
{
    name_ = name;
    salary_ = salary;
}
```

Constructeur par paramètres

- Construction par paramètres d'un objet:

```
Employee marcel = Employee("marcel",50000);
```

Construit un objet de type Employee en fournissant des valeurs initiales à ses attributs

```
Employee marcel("Marcel",50000);
```

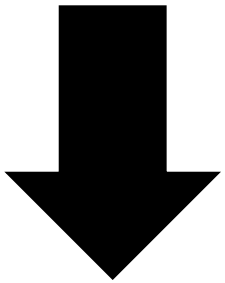
Forme abrégée de l'initialisation par paramètres

Constructeur par paramètres avec valeurs par défaut

- Il est possible de définir un constructeur par paramètres qui admet des valeurs par défaut
- Les valeurs par défaut seront alors explicitées dans la définition de la classe **seulement**
- L'implémentation reste la même que celle du constructeur par paramètres standards
- Ce type de constructeur permet de rassembler différentes surcharges de constructeur en une seule méthode

Constructeur par paramètres avec valeurs par défaut

```
class Employee
{
public:
   Employee();
   Employee(string name);
   Employee(string name, double salary);
    ...
};
```



```
class Employee
{
public:
   Employee(string name = “unknown”, double salary = 0.0);
    ...
};
```

C’est un constructeur par paramètres qui agit aussi comme constructeur par défaut

Destructeur (suite)

- Un destructeur est une méthode qui est appelée lorsqu'un objet est **détruit**
- Le destructeur est défini en lui donnant le même nom que la classe précédé du tilde ~ :

```
Employee::~Employee()
{
    // Rien à faire dans ce cas-ci
}
```

Quand un destructeur est-il appelé?

- Si l'objet est une variable locale à une fonction, il sera **détruit à la sortie de cette fonction**
- Si l'objet est déclaré dans le main(), il sera **détruit à la sortie du programme**
- Si l'objet est dynamique, le destructeur est appelé lorsqu'on exécute **`delete`** sur cet objet

Programmation orientée objet

Liste d'initialisation

Samuel Kadoury

Initialisation des attributs par défaut

```
class Employee
{
public:
   Employee() {}
   Employee(string name, double salary){
       name_ = name;
       salary_ = salary;
   }

private:
   string name_ = "unknown";
   double prime_ = 0.05;
   double salary_ = (1+prime_)*30000.0;
};
```

vide

Problème: Une double initialisation est faite et qui est en lien avec l'ordre de construction d'un objet

Ordre de construction d'un objet

```
class Employee
{
public:
    Employee(string name, double salary){
        name_ = name;
        salary_ = salary;
    }

private:
    string name_ = "unknown";
    double salary_ = 0.0;
};
```

```
int main(){
    Employee bob("Bob",
                 20000.0);
}
```

Les numéros indiquent l'ordre d'exécution des instructions lors de la construction de l'objet **bob**

Liste d'initialisation

```
class Employee
{
public:
   Employee(string name, double salary): salary_(salary),
   name_(name) { }

private:
   string name_ = "unknown";
   double salary_ = 0.0;
};
```

2 5 3 4

La liste d'initialisation permet de spécifier quel constructeur sera appelé lors de la construction des attributs et donc d'éviter la double initialisation des attributs

```
int main(){
   Employee bob("Bob",
                20000.0);
}
```

1

Délégation de constructeur

```
class Employee
{
public:
   Employee(): name_(“unknown”), salary_(0.0) {}
   Employee(string name, double salary): salary_(salary),
   name_(name) {}

private:
   string name_;
   double salary_;
};
```

Problème: Il y a une redondance au niveau de l’implémentation des constructeurs qui peut être résolue grâce à la delegation de constructeur

Délégation de constructeur

```
class Employee{
public:
   Employee(): Employe("unknown", 0.0) {}
   Employee(string name, double salary): salary_(salary),
   name_(name) {}

private:
   string name_;
   double salary_;
};
```

2 3 6 4 5

Le constructeur par défaut de la classe Employee fait maintenant appel au constructeur par paramètres afin de construire l'objet

```
int main(){
   Employee bob;
}
```

1