Programmation orientée objet

Méthodes constantes

Samuel Kadoury

Motivation

- Nous avons vu que certaines méthodes ne servent qu'à accéder à des attributs d'un objet
- Pour éviter toute confusion, on aimerait que ces méthodes ne puissent effectuer aucune modification sur les attributs de l'objet

Implémentation d'une fonction d'accès

- Pour assurer que la méthode ne modifie aucun des attributs de l'objet, on ajoute le mot clé **`const`** après l'en-tête de la fonction:
  - dans la définition de la classe
  - dans l'implémentation de la fonction

Exemple de fonction d'accès

```
string Employee::getName() const
{
    return name_;
}
```

Const pour les getters

Le compilateur retournera un message d'erreur si la fonction contient une instruction qui tente de modifier la valeur d'un attribut de l'objet **ou si la fonction utilise une méthode qui n'a pas été déclarée constante**.

Exemple de code erroné avec const

```
string Employee::getName() const
{
    name_ = "pierre";
    return name_;
}
```

On n'a pas le droit de modifier la valeur d'un attribut de l'objet.

Autre exemple de code erroné avec const

```
class Employee
{
public:
  Employee(string name =
   "unknown", double salary
   = 0);
   double getSalary();
   string getName();
   ...
   void print() const;

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

Constructeur défaut par parametre

```
string Employee::getName()
{
   return name_;
}

double Employee::getSalary()
{
   return salary_;
}

void Employee::print() const
{
   cout << getName() << endl;
   cout << getSalary() << endl;
}
```

Erreur! Pourquoi?

On peut pas utiliser des méthodes qui ne sont pas const.

Autre exemple de code erroné avec const

```
class Employee
{
public:
   Employee(string name =
    "unknown", double salary
    = 0);
   double getSalary()const;
   string getName()const;
   ...
   void print() const;

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

```
string Employee::getName()const
{
    return name_;
}

double Employee::getSalary()const
{
    return salary_;
}

void Employee::print() const
{
    cout << getName() << endl;
    cout << getSalary() << endl;
}
```

Ajout du « const » pour pouvoir écrire le code

Encore un exemple de code erroné avec const

```
class Employee
{
   ...
   void print();
   ...
};
```

```
class Company
{
public:
   ...
   void print() const;
private:
   ...
   Employee president_;
};
```

```
void Company::print() const
{
   ...
   president_.print();
   ...
}
```

On fait appel à une méthode qui n'est pas constante.

Erreur!!
Pourquoi?

Encore un exemple de code erroné av

```
class Employee
{
   ...
   void print();
   ...
};
```

Cette méthode n'est pas déclarée constante.

```
class Company
{
public:
   ...
   void print() const;
private:
   ...
   Employee president_;
};
```

Cette méthode n'a pas le droit de modifier un attribut.

```
void Company::print() const
{
   ...
   president_.print();
   ...
}
```

Puisque la méthode `print` n'est pas déclarée constante, elle pourrait donc modifier l'attribut `president` (même si elle ne le fait pas en pratique). Le compilateur refusera donc de compiler ce code.

Encore un exemple de code erroné avec const

```
class Employee
{
   ...
   void print()const;
    ...
};
```

```
class Company
{
public:
   ...
   void print() const;
private:
    ...
    Employee president_;
};
```

```
void Company::print() const
{
    ...
    president_.print();
    ...
}
```

Le mot clé "**const**" permet de rendre l'état d'une variable constant. Ainsi, lorsqu'une variable est déclarée constante, on ne peut pas changer sa valeur (modifier son état). De plus, si la variable est un objet, on ne peut pas manipuler celle-ci au travers de méthodes qui ne sont pas constantes.
Une méthode constante est une fonction membre qui ne peut pas modifier l'état courant de l'objet. Plus spécifiquement, lorsqu'une méthode est constante, l'objet courant sur lequel elle est appelée est aussi constant. Ainsi, son implémentation ne peut pas comprendre des instructions qui :

1. Modifient l'état de l'objet courant (les attributs de l'objet courant sont constants)
2. Font appel à des méthodes définies dans la classe de l'objet courant qui ne sont pas constantes

Si une méthode déclarée constante tentait de faire l'une ou l'autre de ces actions, le compilateur retournerait un message d'erreur.
Lorsqu'une méthode ne fait aucune modification sur l'objet courant, elle est déclarée comme constante. Un bon exemple de ce type de méthodes est **les fonctions d'accès** (ou getters). Pour déclarer une méthode comme constante, on doit ajouter le mot clé "**const**" après l'en-tête de la méthode dans :

1. La définition de la classe
2. L'implémentation de la méthode

Tous les getters seraient alors des méthodes constantes

Prenons la définition de la classe Employee vu précédemment :

```
class Employee {
public:
  Employee(string name = "unknown", double   salary = 0.0);
  string getName();
  double getSalary();

  void setName(string name);
  void setSalary(double salary);
private:
  string name_;
  double salary_;
};
```

Afin de rendre les fonctions d'accès constantes, on doit ajouter le mot clé "**const**" dans leur définition :

```
class Employee {
public:
        Employee(string name = "unknown", double salary = 0.0);
        string getName() const;
        double getSalary() const;

        void setName(string name);
        void setSalary(double salary);
private:
        string name_;
        double salary_;
};
```

Une fonction globale peut être déclarée constante? FAUX
La fonction globale n'est pas liée à un objet tandis qu'une méthode oui. C'est à cause de ceci que la fonction globale ne peut pas être constante.

Ainsi que dans leur implémentation :

```
string Employee::getName() const {
        return name_;
}

double Employee::getSalary() const {
        return salary_;
}
```

Programmation orientée objet

Passage de paramètres

Samuel Kadoury

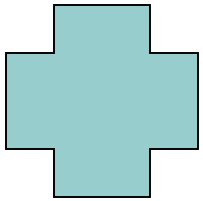
Deux méthodes de passage de paramètre

- ***Passage par valeur:*** on fait une copie du paramètre et c'est cette copie qui sera utilisée à l'intérieur de la fonction
- ***Passage par référence:*** on passe une référence à une entité, c'est-à-dire que l'entité passée en paramètre est manipulée directement, mais sous un autre nom

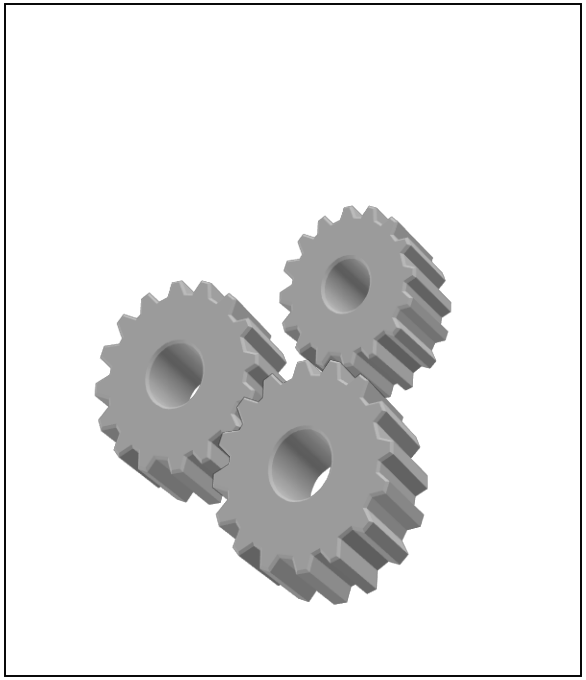
Passage par valeur

FONCTION

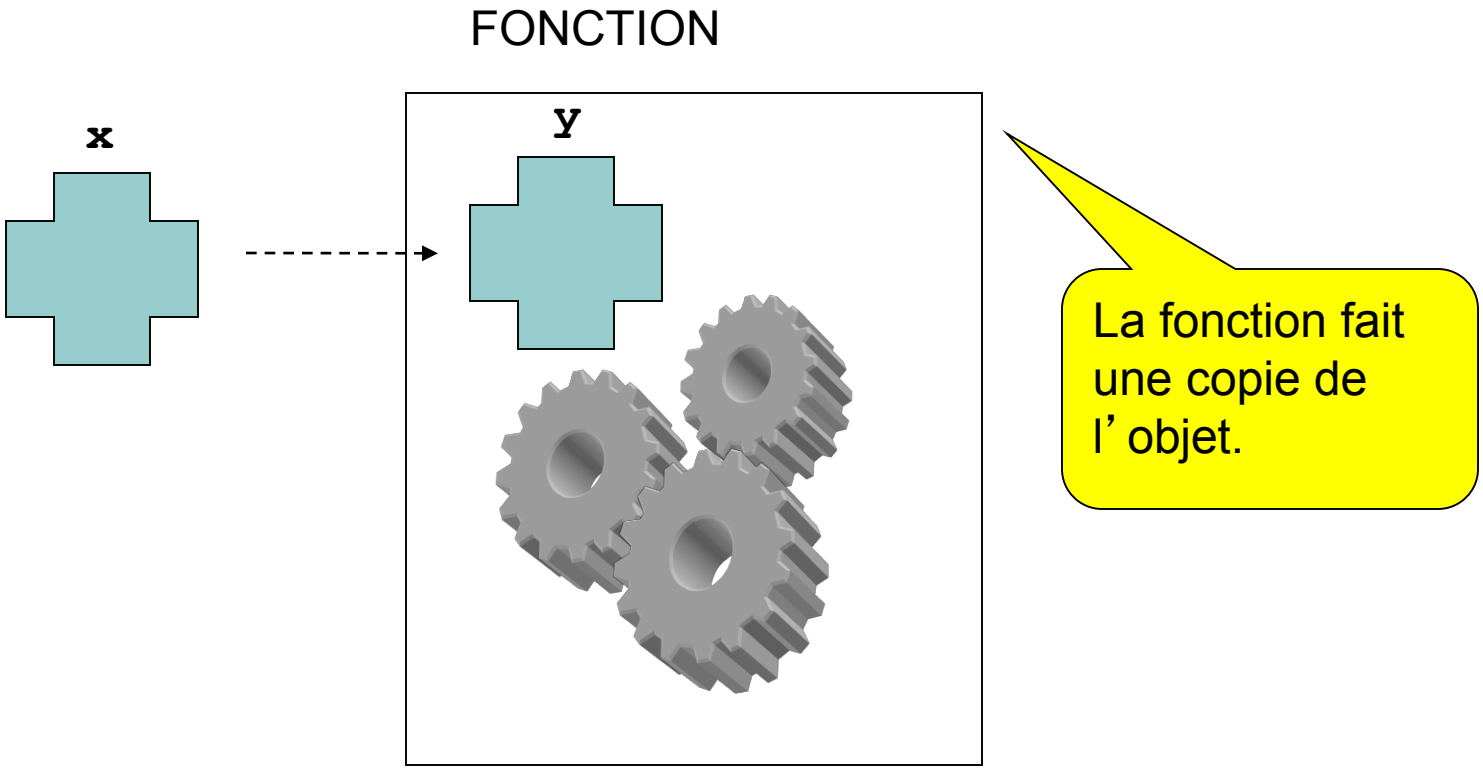
x



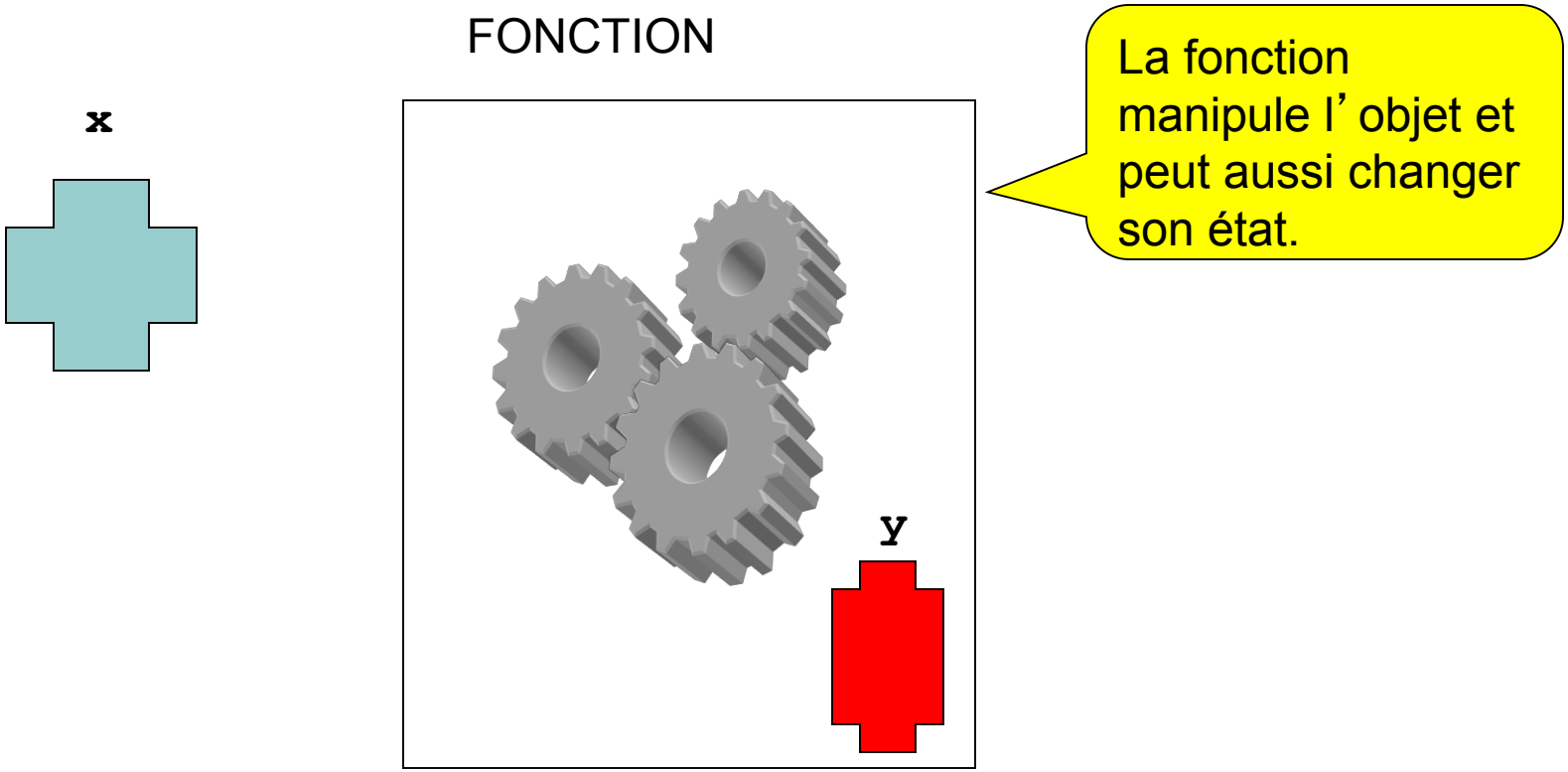
Un objet est passé en paramètre.



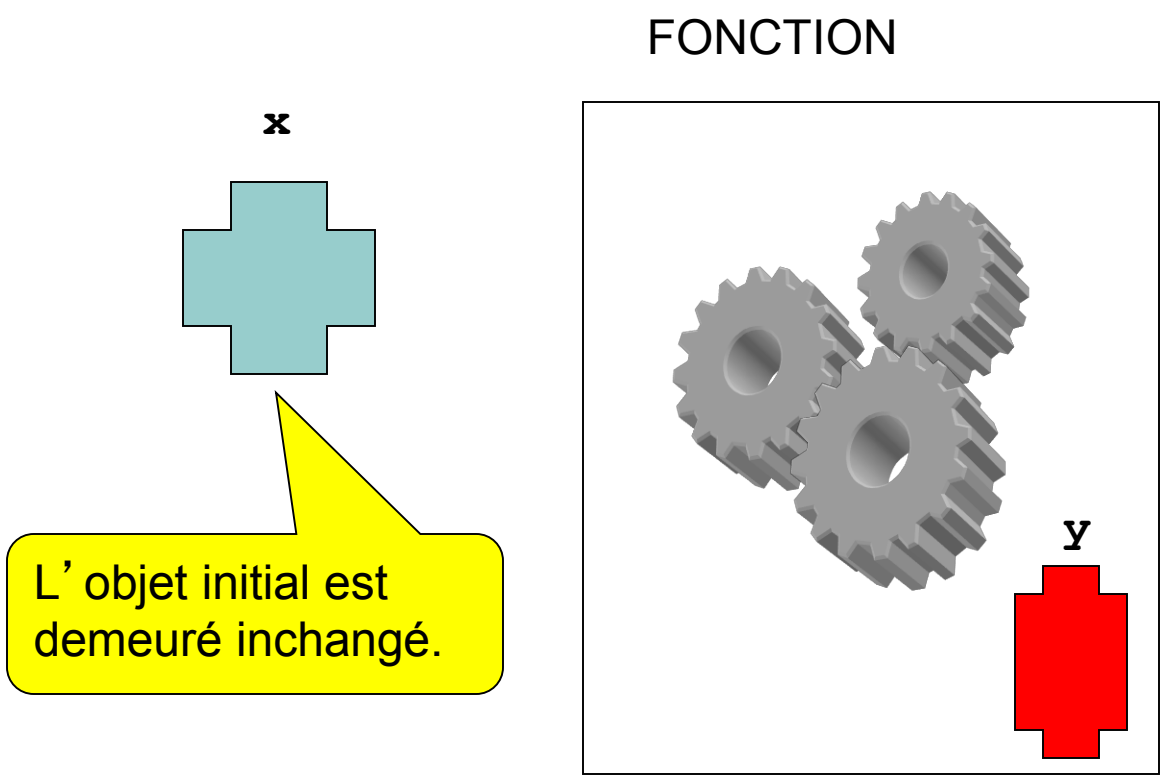
Passage par valeur



Passage par valeur



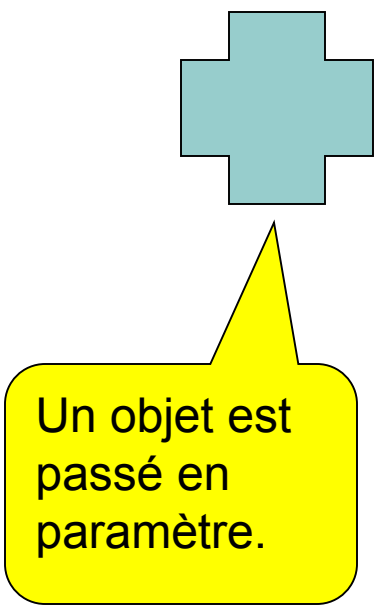
Passage par valeur

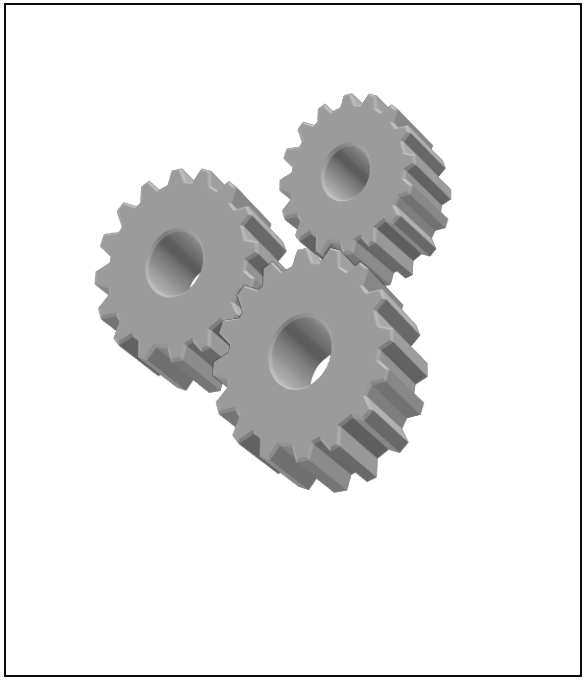


Passage par référence

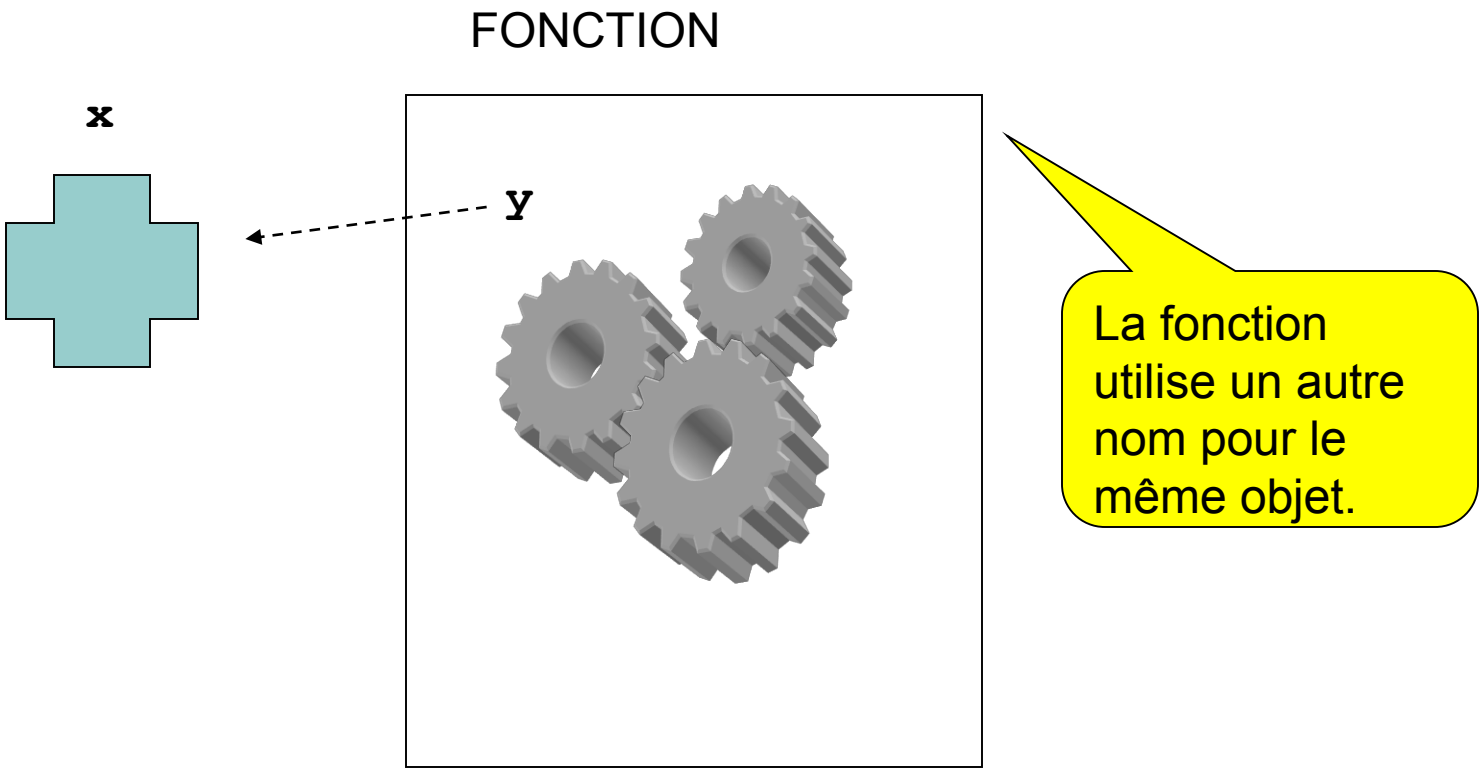
FONCTION

x

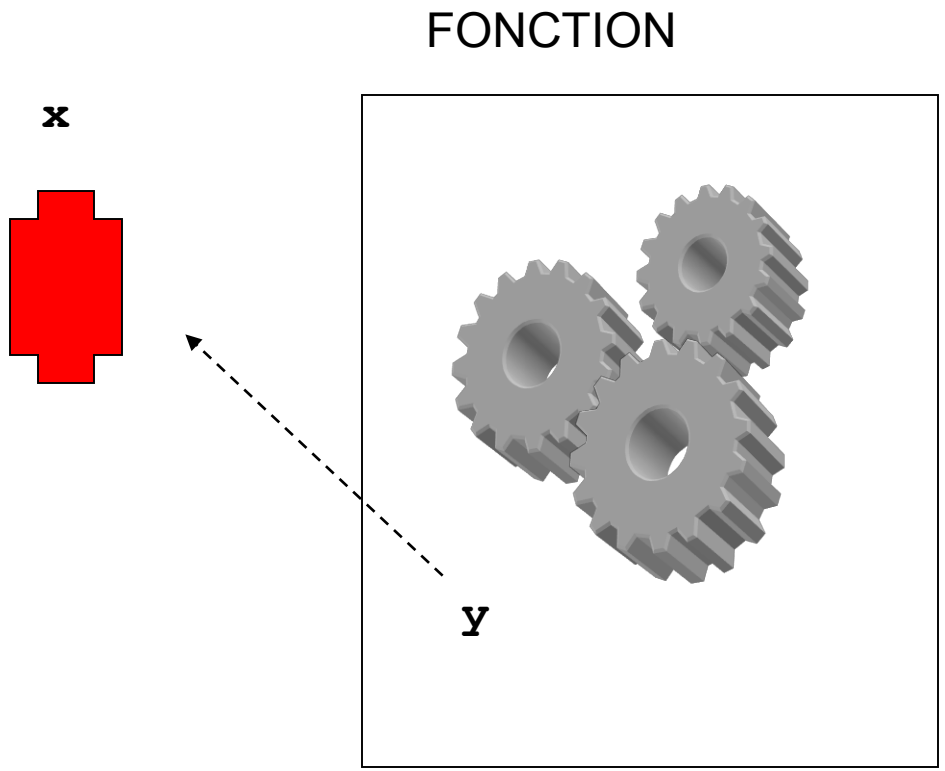




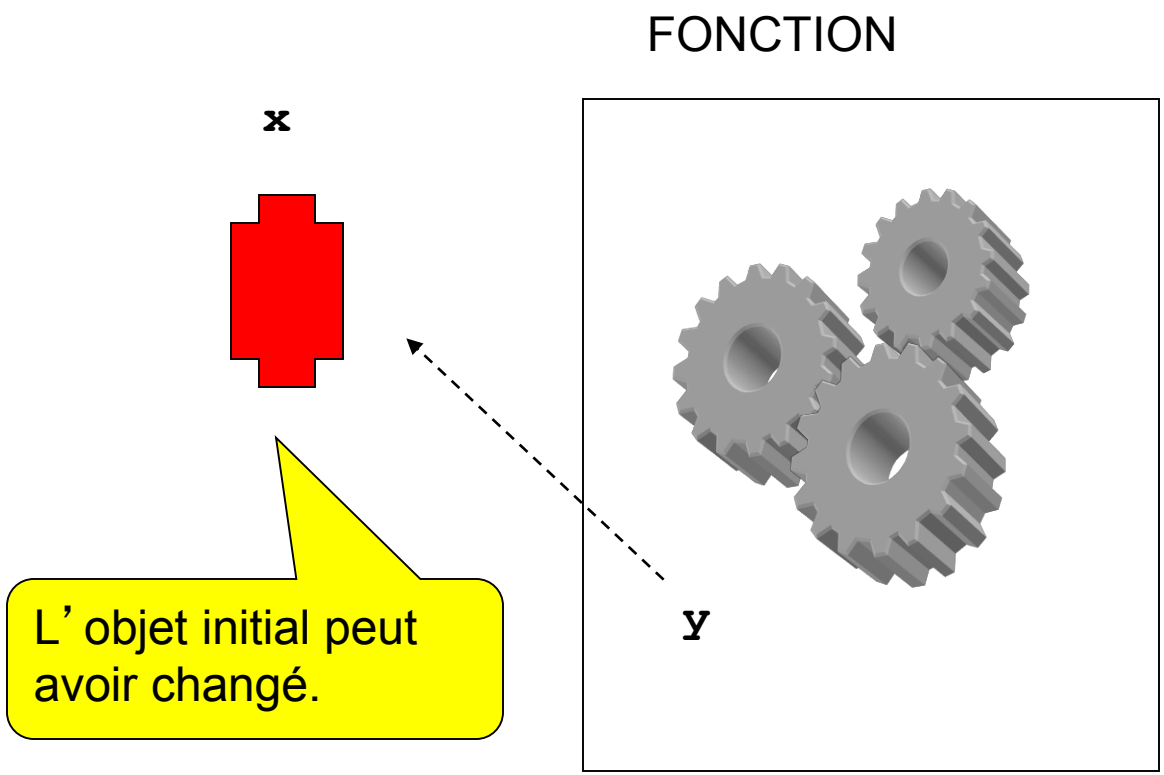
Passage par référence



Passage par référence



Passage par référence



Exemple problématique

```
void increase(Employee employe, double percentage)
{
   double newSalary= employe.getSalary() *
                                (1 + percentage/100);
   employe.setSalary(newSalary);
}

int main()
{
  Employee michel("Michel",100);
  increase(michel,5);
  cout << michel.getSalary();
  ...
}
```

Désolé, mais le salaire n'a pas changé!

Employe*& ?

Exemple corrigé

```
void increase(Employee& employe, double percentage)
{
   double newSalary= employe.getSalary() *
                                (1 + percentage/100);
   employe.setSalary(newSalary);
}

int main()
{
  Employee michel("Michel",100);
  increase(michel,5);
  cout << michel.getSalary();
  ...
}
```

employe est une référence au même objet que celui contenu dans la variable **michel.**

Le salaire aura finalement été augmenté!

Référence constante

- Souvent, on passe un objet par référence non pas parce qu'on veut le modifier, mais plutôt parce qu'on veut **éviter une copie qui est coûteuse**
- Pour éviter que cet objet soit modifié, on utilisera alors une référence constante

Référence constante (exemple)

```
void printCompany(const Company& c)
{
    cout << "Company " << c.getName();
    if (c.hasEmployees()) {
        cout << " has " << c.getNumberEmployees() << " employee(s):"
                                                  << endl;
        for (int i = 0; i < c.getNumberEmployees(); i++) {
            cout << " - Employee " << (i+1) << ": " <<
            c.getEmployeeByPos(i).getName() << endl;
        }
    } else {
        cout << " doesn't have any employee
    }
}
```

Le compilateur permettra seulement l'utilisation de méthodes qui ont été déclarées const.