

# Ch. 8 - Surcharge d'opérateur

## Langage C / C++

R. Absil

Haute École Bruxelles-Brabant  
École supérieure d'Informatique



6 octobre 2021

# Table des matières

- 1 Introduction
- 2 Contraintes
- 3 Surcharge d'opérateur
- 4 Surcharges diverses
- 5 Objets fonctions
- 6 Allocations dynamiques
- 7 Littéraux définis par l'utilisateur

# Table des matières

- 1 Introduction
- 2 Contraintes
- 3 Surcharge d'opérateur
- 4 Surcharges diverses
- 5 Objets fonctions
- 6 Allocations dynamiques
- 7 Littéraux définis par l'utilisateur

# Table des matières

- 1 Introduction
- 2 Contraintes
- 3 Surcharge d'opérateur
- 4 Surcharges diverses
- 5 Objets fonctions
- 6 Allocations dynamiques
- 7 Littéraux définis par l'utilisateur

# Table des matières

- 1 Introduction
- 2 Contraintes
- 3 Surcharge d'opérateur
- 4 Surcharges diverses
- 5 Objets fonctions
- 6 Allocations dynamiques
- 7 Littéraux définis par l'utilisateur

# Table des matières

- 1 Introduction
- 2 Contraintes
- 3 Surcharge d'opérateur
- 4 Surcharges diverses
- 5 Objets fonctions
- 6 Allocations dynamiques
- 7 Littéraux définis par l'utilisateur

# Table des matières

- 1 Introduction
- 2 Contraintes
- 3 Surcharge d'opérateur
- 4 Surcharges diverses
- 5 Objets fonctions
- 6 Allocations dynamiques
- 7 Littéraux définis par l'utilisateur

# Table des matières

- 1 Introduction
- 2 Contraintes
- 3 Surcharge d'opérateur
- 4 Surcharges diverses
- 5 Objets fonctions
- 6 Allocations dynamiques
- 7 Littéraux définis par l'utilisateur



# Introduction

# Overview (1/2)

- C++ **autorise la surdéfinition de fonctions**
  - Fonctions membres
  - Fonctions indépendantes
- Les opérateurs sont des fonctions comme les autres

## Idée

- Surdéfinir certains opérateurs
- $a + b$  doit être interprété correctement si  $a$  et  $b$  sont des vecteurs, par exemple
- Ce comportement existe déjà au sein du langage avec +
  - Addition entière
  - Addition flottante

# Overview (1/2)

- C++ autorise la surdéfinition de fonctions
  - Fonctions membres
    - Fonctions indépendantes
  - Les opérateurs sont des fonctions comme les autres

## Idée

- Surdéfinir certains opérateurs
- $a + b$  doit être interprété correctement si  $a$  et  $b$  sont des vecteurs, par exemple
- Ce comportement existe déjà au sein du langage avec +
  - Addition entière
  - Addition flottante

# Overview (1/2)

- C++ autorise la surdéfinition de fonctions
  - Fonctions membres
  - Fonctions indépendantes
- Les opérateurs sont des fonctions comme les autres

## Idée

- Surdéfinir certains opérateurs
- $a + b$  doit être interprété correctement si  $a$  et  $b$  sont des vecteurs, par exemple
- Ce comportement existe déjà au sein du langage avec +
  - Addition entière
  - Addition flottante

# Overview (1/2)

- C++ autorise la surdéfinition de fonctions
  - Fonctions membres
  - Fonctions indépendantes
- Les opérateurs sont des fonctions comme les autres

## Idée

- Surdéfinir certains opérateurs
- $a + b$  doit être interprété correctement si  $a$  et  $b$  sont des vecteurs, par exemple
- Ce comportement existe déjà au sein du langage avec +
  - Addition entière
  - Addition flottante

# Overview (1/2)

- C++ autorise la surdéfinition de fonctions
  - Fonctions membres
  - Fonctions indépendantes
- Les opérateurs sont des fonctions comme les autres

## Idée

- Surdéfinir certains opérateurs
- $a + b$  doit être interprété correctement si  $a$  et  $b$  sont des vecteurs, par exemple
- Ce comportement existe déjà au sein du langage avec +
  - Addition entière
  - Addition flottante

# Overview (1/2)

- C++ autorise la surdéfinition de fonctions
  - Fonctions membres
  - Fonctions indépendantes
- Les opérateurs sont des fonctions comme les autres

## Idée

- Surdéfinir certains opérateurs
  - $a + b$  doit être interprété correctement si  $a$  et  $b$  sont des vecteurs, par exemple
- Ce comportement existe déjà au sein du langage avec +
  - Addition entière
  - Addition flottante

# Overview (1/2)

- C++ autorise la surdéfinition de fonctions
  - Fonctions membres
  - Fonctions indépendantes
- Les opérateurs sont des fonctions comme les autres

## Idée

- Surdéfinir certains opérateurs
  - $a + b$  doit être interprété correctement si  $a$  et  $b$  sont des vecteurs, par exemple
- 
- Ce comportement existe déjà au sein du langage avec +
    - Addition entière
    - Addition flottante



# Overview (1/2)

- C++ autorise la surdéfinition de fonctions
  - Fonctions membres
  - Fonctions indépendantes
- Les opérateurs sont des fonctions comme les autres

## Idée

- Surdéfinir certains opérateurs
- $a + b$  doit être interprété correctement si  $a$  et  $b$  sont des vecteurs, par exemple
- Ce comportement existe déjà au sein du langage avec +
  - Addition entière
  - Addition flottante

# Overview (1/2)

- C++ autorise la surdéfinition de fonctions
  - Fonctions membres
  - Fonctions indépendantes
- Les opérateurs sont des fonctions comme les autres

## Idée

- Surdéfinir certains opérateurs
- $a + b$  doit être interprété correctement si  $a$  et  $b$  sont des vecteurs, par exemple
- Ce comportement existe déjà au sein du langage avec +
  - Addition entière
  - Addition flottante

# Overview (1/2)

- C++ autorise la surdéfinition de fonctions
  - Fonctions membres
  - Fonctions indépendantes
- Les opérateurs sont des fonctions comme les autres

## Idée

- Surdéfinir certains opérateurs
- $a + b$  doit être interprété correctement si  $a$  et  $b$  sont des vecteurs, par exemple
- Ce comportement existe déjà au sein du langage avec +
  - Addition entière
  - Addition flottante

## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle

- Autre exemple : «

- Décalage de bits
- Impression console

### Exemple

- `int a = add(b,c);`
- `int a = b.add(c);`
- `int a = b + c;`

- Parfois, la surcharge d'opérateur est indispensable

## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle

- Autre exemple : «
  - Décalage de bits
  - Impression console

### Exemple

- `int a = add(b,c);`
- `int a = b.add(c);`
- `int a = b + c;`

- Parfois, la surcharge d'opérateur est indispensable

## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle

- Autre exemple : «
  - Décalage de bits
  - Impression console

### Exemple

- `int a = add(b,c);`
- `int a = b.add(c);`
- `int a = b + c;`

- Parfois, la surcharge d'opérateur est indispensable

## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle
- Autre exemple : «
  - Décalage de bits
  - Impression console

### Exemple

- `int a = add(b,c);`
- `int a = b.add(c);`
- `int a = b + c;`

- Parfois, la surcharge d'opérateur est indispensable

## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle
- Autre exemple : «
  - Décalage de bits
  - Impression console

### Exemple

- `int a = add(b,c);`
- `int a = b.add(c);`
- `int a = b + c;`

- Parfois, la surcharge d'opérateur est indispensable



## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle
- Autre exemple : «
  - Décalage de bits
  - Impression console

### Exemple

- `int a = add(b,c);`
- `int a = b.add(c);`
- `int a = b + c;`

- Parfois, la surcharge d'opérateur est indispensable

## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle
- Autre exemple : «
  - Décalage de bits
  - Impression console

### Exemple

- `int a = add(b,c);`
- `int a = b.add(c);`
- `int a = b + c;`

- Parfois, la surcharge d'opérateur est indispensable

## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle
- Autre exemple : «
  - Décalage de bits
  - Impression console

### Exemple

- `int a = add(b,c);`
- `int a = b.add(c);`
- `int a = b + c;`

- Parfois, la surcharge d'opérateur est indispensable

## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle
- Autre exemple : «
  - Décalage de bits
  - Impression console

### Exemple

```
■ int a = add(b, c);  
■ int a = b.add(c);  
■ int a = b + c;
```

- Parfois, la surcharge d'opérateur est indispensable

## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle
- Autre exemple : «
  - Décalage de bits
  - Impression console

### Exemple

```
■ int a = add(b, c);  
■ int a = b.add(c);  
■ int a = b + c;
```

- Parfois, la surcharge d'opérateur est indispensable

## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle
- Autre exemple : «
  - Décalage de bits
  - Impression console

### Exemple

```
■ int a = add(b, c);  
■ int a = b.add(c);  
■ int a = b + c;
```

- Parfois, la surcharge d'opérateur est indispensable

## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle
- Autre exemple : «
  - Décalage de bits
  - Impression console

### Exemple

```
■ int a = add(b, c);  
■ int a = b.add(c);  
■ int a = b + c;
```

■ Parfois, la surcharge d'opérateur est indispensable

## Overview (2/2)

- Autre exemple :  $*$ 
  - Multiplication
  - Indirection
- La surcharge d'opérateur permet de donner de nouvelles interprétations d'opérateurs *existants*
- Syntaxe opératoire « plus claire » que syntaxe fonctionnelle
- Autre exemple : «
  - Décalage de bits
  - Impression console

### Exemple

```
■ int a = add(b, c);  
■ int a = b.add(c);  
■ int a = b + c;
```

- Parfois, la surcharge d'opérateur est indispensable



# Contraintes

# Règles de base (1/2)

## Idée

- On ne peut pas changer la grammaire du C++
- Impossible de définir un nouvel opérateur
  - $a \$ b$  est interdit
  - $a]b[$  est interdit
- L'arité doit être respectée
  - $a ++ b$  est interdit
- Les priorités, associativités ne peuvent être changées
  - $i + j * k$  est toujours interprété comme  $i + (j * k)$
  - $i + j + k$  est toujours interprété comme  $(i + j) + k$
- Pas de commutativité par défaut
  - Surdéfinir  $+$  entre une classe A et une classe B ne surdéfinit pas  $+$  entre une classe B et une classe A

# Règles de base (1/2)

## Idée

- On ne peut pas changer la grammaire du C++
- Impossible de définir un nouvel opérateur
  - $a \$ b$  est interdit
  - $a]b[$  est interdit
- L'arité doit être respectée
  - $a ++ b$  est interdit
- Les priorités, associativités ne peuvent être changées
  - $i + j * k$  est toujours interprété comme  $i + (j * k)$
  - $i + j + k$  est toujours interprété comme  $(i + j) + k$
- Pas de commutativité par défaut
  - Surdéfinir  $+$  entre une classe A et une classe B ne surdéfinit pas  $+$  entre une classe B et une classe A

# Règles de base (1/2)

## Idée

- On ne peut pas changer la grammaire du C++
- Impossible de définir un nouvel opérateur
  - `a § b` est interdit
  - `a]b[` est interdit
- L'arité doit être respectée
  - `a ++ b` est interdit
- Les priorités, associativités ne peuvent être changées
  - `i + j * k` est toujours interprété comme `i + (j * k)`
  - `i + j + k` est toujours interprété comme `(i + j) + k`
- Pas de commutativité par défaut
  - Surdéfinir `+` entre une classe A et une classe B ne surdéfinit pas `+` entre une classe B et une classe A

# Règles de base (1/2)

## Idée

- On ne peut pas changer la grammaire du C++
- Impossible de définir un nouvel opérateur
  - $a \S b$  est interdit
  - $a]b[$  est interdit
- L'arité doit être respectée
  - $a ++ b$  est interdit
- Les priorités, associativités ne peuvent être changées
  - $i + j * k$  est toujours interprété comme  $i + (j * k)$
  - $i + j + k$  est toujours interprété comme  $(i + j) + k$
- Pas de commutativité par défaut
  - Surdéfinir  $+$  entre une classe A et une classe B ne surdéfinit pas  $+$  entre une classe B et une classe A

# Règles de base (1/2)

## Idée

- On ne peut pas changer la grammaire du C++
- Impossible de définir un nouvel opérateur
  - $a \S b$  est interdit
  - $a]b[$  est interdit
- L'arité doit être respectée
  - $a ++ b$  est interdit
- Les priorités, associativités ne peuvent être changées
  - $i + j * k$  est toujours interprété comme  $i + (j * k)$
  - $i + j + k$  est toujours interprété comme  $(i + j) + k$
- Pas de commutativité par défaut
  - Surdéfinir  $+$  entre une classe A et une classe B ne surdéfinit pas  $+$  entre une classe B et une classe A

# Règles de base (1/2)

## Idée

- On ne peut pas changer la grammaire du C++
- Impossible de définir un nouvel opérateur
  - $a \ \$ \ b$  est interdit
  - $a]b[$  est interdit
- L'arité doit être respectée
  - $a \ ++ \ b$  est interdit
- Les priorités, associativités ne peuvent être changées
  - $i + j * k$  est toujours interprété comme  $i + (j * k)$
  - $i + j + k$  est toujours interprété comme  $(i + j) + k$
- Pas de commutativité par défaut
  - Surdéfinir  $+$  entre une classe A et une classe B ne surdéfinit pas  $+$  entre une classe B et une classe A

# Règles de base (1/2)

## Idée

- On ne peut pas changer la grammaire du C++
- Impossible de définir un nouvel opérateur
  - $a \ \$ \ b$  est interdit
  - $a]b[$  est interdit
- L'arité doit être respectée
  - $a \ ++ \ b$  est interdit
- Les priorités, associativités ne peuvent être changées
  - $i + j * k$  est toujours interprété comme  $i + (j * k)$
  - $i + j + k$  est toujours interprété comme  $(i + j) + k$
- Pas de commutativité par défaut
  - Surdéfinir  $+$  entre une classe A et une classe B ne surdéfinit pas  $+$  entre une classe B et une classe A



# Règles de base (1/2)

## Idée

- On ne peut pas changer la grammaire du C++
- Impossible de définir un nouvel opérateur
  - $a \S b$  est interdit
  - $a]b[$  est interdit
- L'arité doit être respectée
  - $a ++ b$  est interdit
- Les priorités, associativités ne peuvent être changées
  - $i + j * k$  est toujours interprété comme  $i + (j * k)$
  - $i + j + k$  est toujours interprété comme  $(i + j) + k$
- Pas de commutativité par défaut
  - Surdéfinir  $+$  entre une classe A et une classe B ne surdéfinit pas  $+$  entre une classe B et une classe A

# Règles de base (1/2)

## Idée

- On ne peut pas changer la grammaire du C++
- Impossible de définir un nouvel opérateur
  - $a \S b$  est interdit
  - $a]b[$  est interdit
- L'arité doit être respectée
  - $a ++ b$  est interdit
- Les priorités, associativités ne peuvent être changées
  - $i + j * k$  est toujours interprété comme  $i + (j * k)$
  - $i + j + k$  est toujours interprété comme  $(i + j) + k$
- Pas de commutativité par défaut
  - Surdéfinir  $+$  entre une classe A et une classe B ne surdéfinit pas  $+$  entre une classe B et une classe A

# Règles de base (1/2)

## Idée

- On ne peut pas changer la grammaire du C++
- Impossible de définir un nouvel opérateur
  - $a \S b$  est interdit
  - $a]b[$  est interdit
- L'arité doit être respectée
  - $a ++ b$  est interdit
- Les priorités, associativités ne peuvent être changées
  - $i + j * k$  est toujours interprété comme  $i + (j * k)$
  - $i + j + k$  est toujours interprété comme  $(i + j) + k$
- Pas de commutativité par défaut
  - Surdéfinir  $+$  entre une classe A et une classe B ne surdéfinit pas  $+$  entre une classe B et une classe A

# Règles de base (1/2)

## Idée

- On ne peut pas changer la grammaire du C++
- Impossible de définir un nouvel opérateur
  - $a \S b$  est interdit
  - $a]b[$  est interdit
- L'arité doit être respectée
  - $a ++ b$  est interdit
- Les priorités, associativités ne peuvent être changées
  - $i + j * k$  est toujours interprété comme  $i + (j * k)$
  - $i + j + k$  est toujours interprété comme  $(i + j) + k$
- Pas de commutativité par défaut
  - Surdéfinir  $+$  entre une classe A et une classe B ne surdéfinit pas  $+$  entre une classe B et une classe A

# Règles de base (1/2)

## Idée

- On ne peut pas changer la grammaire du C++
- Impossible de définir un nouvel opérateur
  - $a \S b$  est interdit
  - $a]b[$  est interdit
- L'arité doit être respectée
  - $a ++ b$  est interdit
- Les priorités, associativités ne peuvent être changées
  - $i + j * k$  est toujours interprété comme  $i + (j * k)$
  - $i + j + k$  est toujours interprété comme  $(i + j) + k$
- Pas de commutativité par défaut
  - Surdéfinir  $+$  entre une classe  $A$  et une classe  $B$  ne surdéfinit pas  $+$  entre une classe  $B$  et une classe  $A$

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés



## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `--` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `||` surchargés

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
    - localement pour un type en particulier
    - globalement pour tous les types
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
    - localement pour un type en particulier
    - globalement pour tous les types
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
    - localement pour un type en particulier
    - globalement pour tous les types
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés



## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
    - 1 localement pour un type en particulier
    - 2 globalement pour tous les types
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
    - 1 localement pour un type en particulier
    - 2 globalement pour tous les types
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
    - 1 localement pour un type en particulier
    - 2 globalement pour tous les types
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés

## Règles de base (2/2)

- Pas d'implication de signification
  - Redéfinir `+` et `=` ne redéfinit ni `+=` ni `++`
- Certains opérateurs ont une signification par défaut
  - Ils existent sur tous les types « classe »
  - S'ils ne sont pas redéfinis, ils ont cette signification
  - Exemple : `.`, `=`, `->`, etc.
- Les opérateurs `++` et `-` se comportent différemment
  - Selon qu'ils soient préfixés ou suffixés
- Les opérateurs `new` et `delete` ont des paramètres fixés
  - Nombre de bytes à allouer
  - Mémoire à libérer
  - Peuvent être définis
    - 1 localement pour un type en particulier
    - 2 globalement pour tous les types
- L'évaluation paresseuse ne fonctionne pas pour `&&` et `|` | surchargés

# Nécessité de classe

- Un opérateur redéfini doit comporter au moins un argument de type classe
  - 1 Fonction membre : au moins le paramètre implicite `this`
    - Si opérateur unaire : pas d'autre argument
    - Sinon, paramètre explicite
  - 2 Fonction indépendante : un paramètre explicite pour chaque opérande

## Conséquence

- Impossible de redéfinir un opérateur pour les types de base
- Source : <https://ericniebler.com/2017/02/04/operator-overloading/>

# Nécessité de classe

- Un opérateur redéfini doit comporter au moins un argument de type classe
  - 1 Fonction membre : au moins le paramètre implicite `this`
    - Si opérateur unaire : pas d'autre argument
    - Sinon, paramètre explicite
  - 2 Fonction indépendante : un paramètre explicite pour chaque opérande

## Conséquence

- Impossible de redéfinir un opérateur pour les types de base

# Nécessité de classe

- Un opérateur redéfini doit comporter au moins un argument de type classe
  - 1 Fonction membre : au moins le paramètre implicite `this`
    - Si opérateur unaire : pas d'autre argument
    - Sinon, paramètre explicite
  - 2 Fonction indépendante : un paramètre explicite pour chaque opérande

## Conséquence

- Impossible de redéfinir un opérateur pour les types de base

# Nécessité de classe

- Un opérateur redéfini doit comporter au moins un argument de type classe
  - 1 Fonction membre : au moins le paramètre implicite `this`
    - Si opérateur unaire : pas d'autre argument
    - Sinon, paramètre explicite
  - 2 Fonction indépendante : un paramètre explicite pour chaque opérande

## Conséquence

- Impossible de redéfinir un opérateur pour les types de base



# Nécessité de classe

- Un opérateur redéfini doit comporter au moins un argument de type classe
  - 1 Fonction membre : au moins le paramètre implicite `this`
    - Si opérateur unaire : pas d'autre argument
    - Sinon, paramètre explicite
  - 2 Fonction indépendante : un paramètre explicite pour chaque opérande

## Conséquence

- Impossible de redéfinir un opérateur pour les types de base

# Nécessité de classe

- Un opérateur redéfini doit comporter au moins un argument de type classe
  - 1 Fonction membre : au moins le paramètre implicite `this`
    - Si opérateur unaire : pas d'autre argument
    - Sinon, paramètre explicite
  - 2 Fonction indépendante : un paramètre explicite pour chaque opérande

## Conséquence

- Impossible de redéfinir un opérateur pour les types de base
  - Sauf `new` et `delete`

# Nécessité de classe

- Un opérateur redéfini doit comporter au moins un argument de type classe
  - 1 Fonction membre : au moins le paramètre implicite `this`
    - Si opérateur unaire : pas d'autre argument
    - Sinon, paramètre explicite
  - 2 Fonction indépendante : un paramètre explicite pour chaque opérande

## Conséquence

- Impossible de redéfinir un opérateur pour les types de base
  - Sauf `new` et `delete`

# Nécessité de classe

- Un opérateur redéfini doit comporter au moins un argument de type classe
  - 1 Fonction membre : au moins le paramètre implicite `this`
    - Si opérateur unaire : pas d'autre argument
    - Sinon, paramètre explicite
  - 2 Fonction indépendante : un paramètre explicite pour chaque opérande

## Conséquence

- Impossible de redéfinir un opérateur pour les types de base
  - Sauf `new` et `delete`

# Bonnes pratiques

- On peut donner n'importe quelle signification à un opérateur
  - Paramètres, retour, corps, etc.

## Hygiène de programmation

- Donner une signification « intuitive »
- On s'attend à ce que  $+$  entre deux vecteurs définisse l'addition, et que ce soit commutatif
- On s'attend à ce que  $*$  préfixé sur un itérateur retourne la donnée pointée

# Bonnes pratiques

- On peut donner n'importe quelle signification à un opérateur
  - Paramètres, retour, corps, etc.

## Hygiène de programmation

- Donner une signification « intuitive »
- On s'attend à ce que  $+$  entre deux vecteurs définisse l'addition, et que ce soit commutatif
- On s'attend à ce que  $*$  préfixé sur un itérateur retourne la donnée pointée

# Bonnes pratiques

- On peut donner n'importe quelle signification à un opérateur
  - Paramètres, retour, corps, etc.

## Hygiène de programmation

- Donner une signification « intuitive »
- On s'attend à ce que  $+$  entre deux vecteurs définisse l'addition, et que ce soit commutatif
- On s'attend à ce que  $*$  préfixé sur un itérateur retourne la donnée pointée

# Bonnes pratiques

- On peut donner n'importe quelle signification à un opérateur
  - Paramètres, retour, corps, etc.

## Hygiène de programmation

- Donner une signification « intuitive »
- On s'attend à ce que  $+$  entre deux vecteurs définisse l'addition, et que ce soit commutatif
- On s'attend à ce que  $*$  préfixé sur un itérateur retourne la donnée pointée



# Bonnes pratiques

- On peut donner n'importe quelle signification à un opérateur
  - Paramètres, retour, corps, etc.

## Hygiène de programmation

- Donner une signification « intuitive »
- On s'attend à ce que  $+$  entre deux vecteurs définisse l'addition, et que ce soit commutatif
- On s'attend à ce que  $*$  préfixé sur un itérateur retourne la donnée pointée

# Bonnes pratiques

- On peut donner n'importe quelle signification à un opérateur
  - Paramètres, retour, corps, etc.

## Hygiène de programmation

- Donner une signification « intuitive »
- On s'attend à ce que  $+$  entre deux vecteurs définisse l'addition, et que ce soit commutatif
- On s'attend à ce que  $*$  préfixé sur un itérateur retourne la donnée pointée

# Surcharge d'opérateur

# Syntaxe

- Utilisation du mot-clé `operator`
- Deux utilisation possibles
  - Fonction membre
  - Fonction indépendante (souvent amie)

Exemple : classe `fraction`

- Multiplication commutative avec `*`

- Ici, l'impression console est effectuée avec une fonction `toString`
  - Mauvaise pratique
  - Bonne pratique : surcharger « (cf. section suivante)

# Syntaxe

- Utilisation du mot-clé `operator`
- Deux utilisations possibles
  - 1 Fonction membre
  - 2 Fonction indépendante (souvent amie)

Exemple : classe `fraction`

- Multiplication commutative avec `*`

- Ici, l'impression console est effectuée avec une fonction `toString`
  - Mauvaise pratique
  - Bonne pratique : surcharger « (cf. section suivante)

# Syntaxe

- Utilisation du mot-clé `operator`
- Deux utilisation possibles
  - 1 Fonction membre
  - 2 Fonction indépendante (souvent amie)

Exemple : classe `fraction`

- Multiplication commutative avec `*`

- Ici, l'impression console est effectuée avec une fonction `toString`
  - Mauvaise pratique
  - Bonne pratique : surcharger « (cf. section suivante)

# Syntaxe

- Utilisation du mot-clé `operator`
- Deux utilisation possibles
  - 1 Fonction membre
  - 2 Fonction indépendante (souvent amie)

Exemple : classe `fraction`

- Multiplication commutative avec `*`

- Ici, l'impression console est effectuée avec une fonction `toString`
  - Mauvaise pratique
  - Bonne pratique : surcharger « (cf. section suivante)

# Syntaxe

- Utilisation du mot-clé `operator`
- Deux utilisation possibles
  - 1 Fonction membre
  - 2 Fonction indépendante (souvent amie)

## Exemple : classe `fraction`

- Multiplication commutative avec `*`
  - Membre
  - Tous les opérandes sont de type `fraction`
- Ici, l'impression console est effectuée avec une fonction `toString`
  - Mauvaise pratique
  - Bonne pratique : surcharger « (cf. section suivante)



# Syntaxe

- Utilisation du mot-clé `operator`
- Deux utilisation possibles
  - 1 Fonction membre
  - 2 Fonction indépendante (souvent amie)

## Exemple : classe `fraction`

- Multiplication commutative avec `*`
  - Membre
  - Tous les opérandes sont de type `fraction`
- Ici, l'impression console est effectuée avec une fonction `toString`
  - Mauvaise pratique
  - Bonne pratique : surcharger « (cf. section suivante)

# Syntaxe

- Utilisation du mot-clé `operator`
- Deux utilisation possibles
  - 1 Fonction membre
  - 2 Fonction indépendante (souvent amie)

## Exemple : classe `fraction`

- Multiplication commutative avec `*`
  - Membre
  - Tous les opérandes sont de type `fraction`
- Ici, l'impression console est effectuée avec une fonction `toString`
  - Mauvaise pratique
  - Bonne pratique : surcharger « (cf. section suivante)

# Syntaxe

- Utilisation du mot-clé `operator`
- Deux utilisation possibles
  - 1 Fonction membre
  - 2 Fonction indépendante (souvent amie)

## Exemple : classe `fraction`

- Multiplication commutative avec `*`
  - Membre
  - Tous les opérandes sont de type `fraction`
- Ici, l'impression console est effectuée avec une fonction `toString`
  - Mauvaise pratique
  - Bonne pratique : surcharger « (cf. section suivante)

# Syntaxe

- Utilisation du mot-clé `operator`
- Deux utilisation possibles
  - 1 Fonction membre
  - 2 Fonction indépendante (souvent amie)

## Exemple : classe `fraction`

- Multiplication commutative avec `*`
  - Membre
  - Tous les opérandes sont de type `fraction`
- Ici, l'impression console est effectuée avec une fonction `toString`
  - Mauvaise pratique
  - Bonne pratique : surcharger « (cf. section suivante)

# Syntaxe

- Utilisation du mot-clé `operator`
- Deux utilisation possibles
  - 1 Fonction membre
  - 2 Fonction indépendante (souvent amie)

## Exemple : classe `fraction`

- Multiplication commutative avec `*`
  - Membre
  - Tous les opérandes sont de type `fraction`
- Ici, l'impression console est effectuée avec une fonction `toString`
  - Mauvaise pratique
  - Bonne pratique : surcharger « (cf. section suivante)

# Syntaxe

- Utilisation du mot-clé `operator`
- Deux utilisation possibles
  - 1 Fonction membre
  - 2 Fonction indépendante (souvent amie)

## Exemple : classe `fraction`

- Multiplication commutative avec `*`
  - Membre
  - Tous les opérandes sont de type `fraction`
- Ici, l'impression console est effectuée avec une fonction `toString`
  - Mauvaise pratique
  - Bonne pratique : surcharger « (cf. section suivante)

# Exemple

## ■ Fichier `fraction.cpp`

```

1  class fraction
2  {
3      unsigned num, denom;
4      bool positive;
5
6      public:
7          fraction(int num = 0, int denom = 1);
8          fraction(unsigned num, unsigned denom, bool positive);
9
10         fraction operator *(fraction f) const; //member
11         //friend fraction operator *(fraction f1, fraction f2); //indep
12     };

```

- La multiplication est définie comme fonction membre
  - Un opérande implicite : `this`
  - Un opérande explicite : `f`
- La multiplication est définie comme fonction indépendante
  - Deux opérandes explicites : `f1` et `f2`
  - Permet de préserver la symétrie avec types primitifs
    - Cf. Ch. 10 dédié aux conversions

# Exemple avec opérateur $\star$ membre

## ■ Fichier `fraction.cpp`

```
1 fraction::fraction(int num, int denom)
2   : num(abs(num)), denom(abs(denom)),
3     positive((num >= 0 && denom >= 0) || (num <= 0 && denom <= 0))
4 {}
5
6 fraction::fraction(unsigned num, unsigned denom, bool positive)
7   : num(num), denom(denom), positive(positive)
8 {}
9
10 fraction fraction::operator *(fraction f) const
11 {
12     return fraction(num * f.num, denom * f.denom, //overflow unsafe, use gcd and lcm
13                     (positive && f.positive) || (!positive && !f.positive));
14 }
```



# Exemple avec opérateur $\star$ indépendant

## ■ Fichier `fraction.cpp`

```
1  class fraction
2  {
3      ...
4
5      public:
6          ...
7
8          friend fraction operator *(fraction f1, fraction f2);
9  };
10
11 fraction operator*(fraction f1, fraction f2)
12 {
13     return fraction(f1.num * f2.num, f1.denom * f2.denom, //overflow unsafe
14                     (f1.positive && f2.positive) || (! f1.positive && !f2.positive));
15 }
```

# Remarques

- Une instruction  $f1 * f2 * f3$ 
  - est évaluée comme  $(f1 * f2) * f3$  (langage)
  - crée des objets temporaires
    - Leur nombre dépend du compilateur
    - Des passages par adresse de temporaires peuvent être « cachés »
- On peut savoir ce que le compilateur fait en réécrivant les constructeurs de copie, par défaut, surcharger l'opérateur &, etc.

## Hygiène de programmation

- Définissez des opérateurs indépendants du compilateur

# Remarques

- Une instruction  $f1 * f2 * f3$ 
  - est évaluée comme  $(f1 * f2) * f3$  (langage)
  - crée des objets temporaires
    - Leur nombre dépend du compilateur
    - Des passages par adresse de temporaires peuvent être « cachés »
- On peut savoir ce que le compilateur fait en réécrivant les constructeurs de copie, par défaut, surcharger l'opérateur &, etc.

## Hygiène de programmation

- Définissez des opérateurs indépendants du compilateur

# Remarques

- Une instruction  $f1 * f2 * f3$ 
  - est évaluée comme  $(f1 * f2) * f3$  (langage)
  - crée des objets temporaires
    - Leur nombre dépend du compilateur
    - Des passages par adresse de temporaires peuvent être « cachés »
- On peut savoir ce que le compilateur fait en réécrivant les constructeurs de copie, par défaut, surcharger l'opérateur &, etc.

## Hygiène de programmation

- Définissez des opérateurs indépendants du compilateur

# Remarques

- Une instruction  $f1 * f2 * f3$ 
  - est évaluée comme  $(f1 * f2) * f3$  (langage)
  - crée des objets temporaires
    - Leur nombre dépend du compilateur
    - Des passages par adresse de temporaires peuvent être « cachés »
- On peut savoir ce que le compilateur fait en réécrivant les constructeurs de copie, par défaut, surcharger l'opérateur &, etc.

## Hygiène de programmation

- Définissez des opérateurs indépendants du compilateur

# Remarques

- Une instruction  $f1 * f2 * f3$ 
  - est évaluée comme  $(f1 * f2) * f3$  (langage)
  - crée des objets temporaires
    - Leur nombre dépend du compilateur
    - Des passages par adresse de temporaires peuvent être « cachés »
- On peut savoir ce que le compilateur fait en réécrivant les constructeurs de copie, par défaut, surcharger l'opérateur &, etc.

## Hygiène de programmation

- Définissez des opérateurs indépendants du compilateur

# Remarques

- Une instruction  $f1 * f2 * f3$ 
  - est évaluée comme  $(f1 * f2) * f3$  (langage)
  - crée des objets temporaires
    - Leur nombre dépend du compilateur
    - Des passages par adresse de temporaires peuvent être « cachés »
- On peut savoir ce que le compilateur fait en réécrivant les constructeurs de copie, par défaut, surcharger l'opérateur  $\&$ , etc.

## Hygiène de programmation

- Définissez des opérateurs indépendants du compilateur

# Remarques

- Une instruction  $f1 * f2 * f3$ 
  - est évaluée comme  $(f1 * f2) * f3$  (langage)
  - crée des objets temporaires
    - Leur nombre dépend du compilateur
    - Des passages par adresse de temporaires peuvent être « cachés »
- On peut savoir ce que le compilateur fait en réécrivant les constructeurs de copie, par défaut, surcharger l'opérateur  $\&$ , etc.

## Hygiène de programmation

- Définissez des opérateurs indépendants du compilateur



# Remarques

- Une instruction  $f1 * f2 * f3$ 
  - est évaluée comme  $(f1 * f2) * f3$  (langage)
  - crée des objets temporaires
    - Leur nombre dépend du compilateur
    - Des passages par adresse de temporaires peuvent être « cachés »
- On peut savoir ce que le compilateur fait en réécrivant les constructeurs de copie, par défaut, surcharger l'opérateur  $\&$ , etc.

## Hygiène de programmation

- Définissez des opérateurs indépendants du compilateur

# Liberté

- Les opérateurs sont des fonctions comme les autres
- Possibilité de
  - transmettre les opérandes par référence
    - `int` et `double` pas possible
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - `new` et `delete` pour les objets
    - `new` et `delete` pour les `string` et `vector`
    - `new` et `delete` pour les `array` et `map`
  - protéger les arguments contre la modification avec `const`
    - `const` ne sert que s'ils sont transmis par référence
  - les rendre constants (`const` en fin de prototype)
  - les rendre `inline`

# Liberté

- Les opérateurs sont des fonctions comme les autres
- Possibilité de
  - transmettre les opérandes par référence
    - Utile si gros objets
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - Overhead si petits objets
    - Gestion mémoire « complexe », cf. Ch. 5
  - protéger les arguments contre la modification avec `const`
    - N'a de sens que s'ils sont transmis par référence
  - les rendre constants (`const` en fin de prototype)
  - les rendre `inline`

# Liberté

- Les opérateurs sont des fonctions comme les autres
- Possibilité de
  - transmettre les opérandes par référence
    - Utile si gros objets
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - Overhead si petits objets
    - Gestion mémoire « complexe », cf. Ch. 5
  - protéger les arguments contre la modification avec `const`
    - N'a de sens que s'ils sont transmis par référence
  - les rendre constants (`const` en fin de prototype)
  - les rendre `inline`

# Liberté

- Les opérateurs sont des fonctions comme les autres
- Possibilité de
  - transmettre les opérandes par référence
    - Utile si gros objets
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - Overhead si petits objets
    - Gestion mémoire « complexe », cf. Ch. 5
  - protéger les arguments contre la modification avec `const`
    - N'a de sens que s'ils sont transmis par référence
  - les rendre constants (`const` en fin de prototype)
  - les rendre `inline`

# Liberté

- Les opérateurs sont des fonctions comme les autres
- Possibilité de
  - transmettre les opérandes par référence
    - Utile si gros objets
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - Overhead si petits objets
    - Gestion mémoire « complexe », cf. Ch. 5
  - protéger les arguments contre la modification avec `const`
    - N'a de sens que s'ils sont transmis par référence
  - les rendre constants (`const` en fin de prototype)
  - les rendre `inline`

# Liberté

- Les opérateurs sont des fonctions comme les autres
- Possibilité de
  - transmettre les opérandes par référence
    - Utile si gros objets
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - Overhead si petits objets
    - Gestion mémoire « complexe », cf. Ch. 5
  - protéger les arguments contre la modification avec `const`
    - N'a de sens que s'ils sont transmis par référence
  - les rendre constants (`const` en fin de prototype)
  - les rendre `inline`

# Liberté

- Les opérateurs sont des fonctions comme les autres
- Possibilité de
  - transmettre les opérandes par référence
    - Utile si gros objets
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - Overhead si petits objets
    - Gestion mémoire « complexe », cf. Ch. 5
  - protéger les arguments contre la modification avec `const`
    - N'a de sens que s'ils sont transmis par référence
  - les rendre constants (`const` en fin de prototype)
  - les rendre `inline`



# Liberté

- Les opérateurs sont des fonctions comme les autres
- Possibilité de
  - transmettre les opérandes par référence
    - Utile si gros objets
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - Overhead si petits objets
    - Gestion mémoire « complexe », cf. Ch. 5
  - protéger les arguments contre la modification avec `const`
    - N'a de sens que s'ils sont transmis par référence
  - les rendre constants (`const` en fin de prototype)
  - les rendre `inline`

# Liberté

- Les opérateurs sont des fonctions comme les autres
- Possibilité de
  - transmettre les opérandes par référence
    - Utile si gros objets
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - Overhead si petits objets
    - Gestion mémoire « complexe », cf. Ch. 5
  - protéger les arguments contre la modification avec `const`
    - N'a de sens que s'ils sont transmis par référence
  - les rendre constants (`const` en fin de prototype)
  - les rendre `inline`

# Liberté

- Les opérateurs sont des fonctions comme les autres
- Possibilité de
  - transmettre les opérandes par référence
    - Utile si gros objets
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - Overhead si petits objets
    - Gestion mémoire « complexe », cf. Ch. 5
  - protéger les arguments contre la modification avec `const`
    - N'a de sens que s'ils sont transmis par référence
  - les rendre constants (`const` en fin de prototype)
  - les rendre `inline`

# Liberté

- Les opérateurs sont des fonctions comme les autres
- Possibilité de
  - transmettre les opérandes par référence
    - Utile si gros objets
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - Overhead si petits objets
    - Gestion mémoire « complexe », cf. Ch. 5
  - protéger les arguments contre la modification avec `const`
    - N'a de sens que s'ils sont transmis par référence
  - les rendre constants (`const` en fin de prototype)
  - les rendre `inline`

# Liberté

- Les opérateurs sont des fonctions comme les autres
- Possibilité de
  - transmettre les opérandes par référence
    - Utile si gros objets
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - Overhead si petits objets
    - Gestion mémoire « complexe », cf. Ch. 5
  - protéger les arguments contre la modification avec `const`
    - N'a de sens que s'ils sont transmis par référence
  - les rendre constants (`const` en fin de prototype)
  - les rendre `inline`

# Surcharges diverses

# Impression console

- Pour l'instant, l'impression console était effectuée via
  - une fonction `print`
    - Problème de couplage I/O
  - une fonction `toString`, avec `string`
    - Inefficace

## En pratique

- Surcharge de `ostream& «(ostream&, const MaClasse& brol)`
- Avec une fonction amie, indépendante
- Comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite autoriser certaines conversions
- On retourne le résultat de l'impression par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie (interdit)

# Impression console

- Pour l'instant, l'impression console était effectuée via
  - une fonction `print`
    - Problème de couplage I/O
  - une fonction `toString`, avec `string`
    - Inefficace

## En pratique

- Surcharge de `ostream& «(ostream&, const MaClasse& brol)`
- Avec une fonction amie, indépendante
- Comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite autoriser certaines conversions
- On retourne le résultat de l'impression par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie (interdit)



# Impression console

- Pour l'instant, l'impression console était effectuée via
  - une fonction `print`
    - Problème de couplage I/O
  - une fonction `toString`, avec `string`
    - Inefficace

## En pratique

- Surcharge de `ostream& «(ostream&, const MaClasse& brol)`
- Avec une fonction amie, indépendante
- Comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite autoriser certaines conversions
- On retourne le résultat de l'impression par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie (interdit)

# Impression console

- Pour l'instant, l'impression console était effectuée via
  - une fonction `print`
    - Problème de couplage I/O
  - une fonction `toString`, avec `string`
    - Inefficace

## En pratique

- Surcharge de `ostream& «(ostream&, const MaClasse& brol)`
- Avec une fonction amie, indépendante
- Comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite autoriser certaines conversions
- On retourne le résultat de l'impression par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie (interdit)

# Impression console

- Pour l'instant, l'impression console était effectuée via
  - une fonction `print`
    - Problème de couplage I/O
  - une fonction `toString`, avec `string`
    - Inefficace

## En pratique

- Surcharge de `ostream& «(ostream&, const MaClasse& brol)`
- Avec une fonction amie, indépendante
- Comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite autoriser certaines conversions
- On retourne le résultat de l'impression par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie (interdit)

# Impression console

- Pour l'instant, l'impression console était effectuée via
  - une fonction `print`
    - Problème de couplage I/O
  - une fonction `toString`, avec `string`
    - Inefficace

## En pratique

- Surcharge de `ostream& «(ostream&, const MaClasse& brol)`
- Avec une fonction amie, indépendante
- Comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite autoriser certaines conversions
- On retourne le résultat de l'impression par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie (interdit)

# Impression console

- Pour l'instant, l'impression console était effectuée via
  - une fonction `print`
    - Problème de couplage I/O
  - une fonction `toString`, avec `string`
    - Inefficace

## En pratique

- **Surcharge de** `ostream& «(ostream&, const MaClasse& brol)`
  - Avec une fonction amie, indépendante
- Comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite autoriser certaines conversions
- On retourne le résultat de l'impression par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie (interdit)

# Impression console

- Pour l'instant, l'impression console était effectuée via
  - une fonction `print`
    - Problème de couplage I/O
  - une fonction `toString`, avec `string`
    - Inefficace

## En pratique

- Surcharge de `ostream& «(ostream&, const MaClasse& brol)`
- Avec une fonction amie, indépendante
- Comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite autoriser certaines conversions
- On retourne le résultat de l'impression par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie (interdit)

# Impression console

- Pour l'instant, l'impression console était effectuée via
  - une fonction `print`
    - Problème de couplage I/O
  - une fonction `toString`, avec `string`
    - Inefficace

## En pratique

- Surcharge de `ostream& «(ostream&, const MaClasse& brol)`
- Avec une fonction amie, indépendante
- Comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite autoriser certaines conversions
- On retourne le résultat de l'impression par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie (interdit)

# Impression console

- Pour l'instant, l'impression console était effectuée via
  - une fonction `print`
    - Problème de couplage I/O
  - une fonction `toString`, avec `string`
    - Inefficace

## En pratique

- Surcharge de `ostream& «(ostream&, const MaClasse& brol)`
- Avec une fonction amie, indépendante
- Comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite autoriser certaines conversions
- On retourne le résultat de l'impression par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie (interdit)



# Impression console

- Pour l'instant, l'impression console était effectuée via
  - une fonction `print`
    - Problème de couplage I/O
  - une fonction `toString`, avec `string`
    - Inefficace

## En pratique

- Surcharge de `ostream& «(ostream&, const MaClasse& brol)`
- Avec une fonction amie, indépendante
- Comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite autoriser certaines conversions
- On retourne le résultat de l'impression par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie (interdit)

# Impression console

- Pour l'instant, l'impression console était effectuée via
  - une fonction `print`
    - Problème de couplage I/O
  - une fonction `toString`, avec `string`
    - Inefficace

## En pratique

- Surcharge de `ostream& «(ostream&, const MaClasse& brol)`
  - Avec une fonction amie, indépendante
- 
- Comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite autoriser certaines conversions
  - On retourne le résultat de l'impression par référence afin
    - de pouvoir traiter les affectations multiples
    - d'éviter d'appeler le constructeur de copie (interdit)

# Exemple

## ■ Fichier `point.cpp`

```
1  class point
2  {
3      double _x, _y;
4
5      public:
6          point(double x = 0, double y = 0) : _x(x), _y(y) {}
7
8          inline double x() const { return _x; }
9          inline double y() const { return _y; }
10
11         friend ostream& operator << (ostream& out, const point& p);
12     };
13
14     ostream& operator << (ostream& out, const point& p)
15     {
16         out << "(" << p._x << ", " << p._y << ")";
17         return out;
18     }
```

# Opérateurs arithmétiques

- Cf. section précédente avec `*` (`fraction.cpp`)

## Membre vs indépendant

- Définir en membre `A::operator +(int a) ;` permet de faire
- En indépendant, cette possibilité existe
- Autre possibilité pour les membres : conversions définies par l'utilisateur (cf. Ch. 10)
- Éviter la redondance (+, +=, symétrie, etc.)
  - Définir une implémentation
  - Les autres sont `inline` et appellent la première

# Opérateurs arithmétiques

- Cf. section précédente avec `*` (`fraction.cpp`)

## Membre vs indépendant

- Définir en membre A `operator +(int a) ;` permet de faire
  - `a + 2`
  - mais pas `2 + a`
- En indépendant, cette possibilité existe
  - Fournir deux implémentations
- Autre possibilité pour les membres : conversions définies par l'utilisateur (cf. Ch. 10)
- Éviter la redondance (+, +=, symétrie, etc.)
  - Définir une implémentation
  - Les autres sont `inline` et appellent la première

# Opérateurs arithmétiques

- Cf. section précédente avec `*` (`fraction.cpp`)

## Membre vs indépendant

- Définir en membre `A operator + (int a) ;` permet de faire
  - `a + 2`
  - mais pas `2 + a`
- En indépendant, cette possibilité existe
  - Fournir deux implémentations
- Autre possibilité pour les membres : conversions définies par l'utilisateur (cf. Ch. 10)
- Éviter la redondance (+, +=, symétrie, etc.)
  - Définir une implémentation
  - Les autres sont `inline` et appellent la première

# Opérateurs arithmétiques

- Cf. section précédente avec `*` (`fraction.cpp`)

## Membre vs indépendant

- Définir en membre `A operator + (int a) ;` permet de faire
  - `a + 2`
  - mais pas `2 + a`
- En indépendant, cette possibilité existe
  - Fournir deux implémentations
- Autre possibilité pour les membres : conversions définies par l'utilisateur (cf. Ch. 10)
- Éviter la redondance (+, +=, symétrie, etc.)
  - Définir une implémentation
  - Les autres sont `inline` et appellent la première

# Opérateurs arithmétiques

- Cf. section précédente avec `*` (`fraction.cpp`)

## Membre vs indépendant

- Définir en membre `A operator + (int a) ;` permet de faire
  - `a + 2`
  - mais pas `2 + a`
- En indépendant, cette possibilité existe
  - Fournir deux implémentations
- Autre possibilité pour les membres : conversions définies par l'utilisateur (cf. Ch. 10)
- Éviter la redondance (+, +=, symétrie, etc.)
  - Définir une implémentation
  - Les autres sont `inline` et appellent la première



# Opérateurs arithmétiques

- Cf. section précédente avec `*` (`fraction.cpp`)

## Membre vs indépendant

- Définir en membre `A operator + (int a) ;` permet de faire
  - `a + 2`
  - mais pas `2 + a`
- En indépendant, cette possibilité existe
  - Fournir deux implémentations
- Autre possibilité pour les membres : conversions définies par l'utilisateur (cf. Ch. 10)
- Éviter la redondance (+, +=, symétrie, etc.)
  - Définir une implémentation
  - Les autres sont `inline` et appellent la première

# Opérateurs arithmétiques

- Cf. section précédente avec `*` (`fraction.cpp`)

## Membre vs indépendant

- Définir en membre `A operator + (int a) ;` permet de faire
  - `a + 2`
  - mais pas `2 + a`
- En indépendant, cette possibilité existe
  - Fournir deux implémentations
- Autre possibilité pour les membres : conversions définies par l'utilisateur (cf. Ch. 10)
- Éviter la redondance (+, +=, symétrie, etc.)
  - Définir une implémentation
  - Les autres sont `inline` et appellent la première

# Opérateurs arithmétiques

- Cf. section précédente avec `*` (`fraction.cpp`)

## Membre vs indépendant

- Définir en membre `A operator + (int a) ;` permet de faire
  - `a + 2`
  - mais pas `2 + a`
- En indépendant, cette possibilité existe
  - Fournir deux implémentations
- Autre possibilité pour les membres : conversions définies par l'utilisateur (cf. Ch. 10)
- Éviter la redondance (+, +=, symétrie, etc.)
  - Définir une implémentation
  - Les autres sont `inline` et appellent la première

# Opérateurs arithmétiques

- Cf. section précédente avec `*` (`fraction.cpp`)

## Membre vs indépendant

- Définir en membre `A operator + (int a) ;` permet de faire
  - `a + 2`
  - mais pas `2 + a`
- En indépendant, cette possibilité existe
  - Fournir deux implémentations
- Autre possibilité pour les membres : conversions définies par l'utilisateur (cf. Ch. 10)
- Éviter la redondance (`+`, `+=`, symétrie, etc.)
  - Définir une implémentation
  - Les autres sont `inline` et appellent la première

# Opérateurs arithmétiques

- Cf. section précédente avec `*` (`fraction.cpp`)

## Membre vs indépendant

- Définir en membre `A operator + (int a) ;` permet de faire
  - `a + 2`
  - mais pas `2 + a`
- En indépendant, cette possibilité existe
  - Fournir deux implémentations
- Autre possibilité pour les membres : conversions définies par l'utilisateur (cf. Ch. 10)
- Éviter la redondance (`+`, `+=`, symétrie, etc.)
  - Définir une implémentation
  - Les autres sont `inline` et appellent la première

# Opérateurs arithmétiques

- Cf. section précédente avec `*` (`fraction.cpp`)

## Membre vs indépendant

- Définir en membre `A operator + (int a) ;` permet de faire
  - `a + 2`
  - mais pas `2 + a`
- En indépendant, cette possibilité existe
  - Fournir deux implémentations
- Autre possibilité pour les membres : conversions définies par l'utilisateur (cf. Ch. 10)
- Éviter la redondance (`+`, `+=`, symétrie, etc.)
  - Définir une implémentation
  - Les autres sont `inline` et appellent la première

# Exemple

## ■ Fichier vector2d.cpp

```

1  class vector2d
2  {
3      double _x, _y;
4
5      public:
6          vector2d(double x = 0, double y = 0) : _x(x), _y(y) {}
7
8          vector2d& operator +=(const vector2d& v)
9          {
10             _x += v._x;
11             _y += v._y;
12
13             return *this;
14         }
15
16         friend vector2d operator + (vector2d v1, const vector2d& v2)
17         {
18             v1 += v2;
19             return v1;
20         }
21
22         friend ostream& operator <<(ostream& out, const vector2d& v)
23         {
24             return out << "(" << v._x << ", " << v._y << ")";
25         }
26     };

```

# Surcharge de []

## ■ Fichier charset.cpp

```
1  class CharSet
2  {
3      vector<pair<char, unsigned> > codes;
4
5      public:
6          void update(char c, unsigned code)
7          {
8              int i = find(c);
9              if (i == -1)
10                 codes.push_back(std::make_pair(c, code));
11             else
12                 codes[i].second = code;
13         }
14
15         unsigned& operator [] (char c)
16         {
17             int i = find(c);
18             if (i == -1)
19                 throw std::out_of_range("Invalide_char");
20             else
21                 return codes[i].second;
22         }
23         ...
24     };
25
```



# Remarques

- Transmettre le retour de l'opérateur par référence permet d'écrire des instructions telles que `set['a'] = 12;`
  - Car c'est une lvalue
- Si l'on veut empêcher ce comportement, mais toujours éviter les copies, on peut retourner un `const`
- Souvent, les deux implémentations sont fournies
  - La variante `const` est appelée sur les objets `const`
- En général, la communauté préfère éviter de créer du code permettant d'utiliser `[] []`
  - Complexe
  - L'opérateur `[]` est unaire
  - Utiliser l'opérateur `()` (arité variable)

# Remarques

- Transmettre le retour de l'opérateur par référence permet d'écrire des instructions telles que `set['a'] = 12;`
  - Car c'est une lvalue
- Si l'on veut empêcher ce comportement, mais toujours éviter les copies, on peut retourner un `const`
- Souvent, les deux implémentations sont fournies
  - La variante `const` est appelée sur les objets `const`
- En général, la communauté préfère éviter de créer du code permettant d'utiliser `[] []`
  - Complexe
  - L'opérateur `[]` est unaire
  - Utiliser l'opérateur `()` (arité variable)

# Remarques

- Transmettre le retour de l'opérateur par référence permet d'écrire des instructions telles que `set['a'] = 12;`
  - Car c'est une lvalue
- Si l'on veut empêcher ce comportement, mais toujours éviter les copies, on peut retourner un `const`
- Souvent, les deux implémentations sont fournies
  - La variante `const` est appelée sur les objets `const`
- En général, la communauté préfère éviter de créer du code permettant d'utiliser `[] []`
  - Complexe
  - L'opérateur `[]` est unaire
  - Utiliser l'opérateur `()` (arité variable)

# Remarques

- Transmettre le retour de l'opérateur par référence permet d'écrire des instructions telles que `set['a'] = 12;`
  - Car c'est une lvalue
- Si l'on veut empêcher ce comportement, mais toujours éviter les copies, on peut retourner un `const`
- Souvent, les deux implémentations sont fournies
  - La variante `const` est appelée sur les objets `const`
- En général, la communauté préfère éviter de créer du code permettant d'utiliser `[] []`
  - Complexe
  - L'opérateur `[]` est unaire
  - Utiliser l'opérateur `()` (arité variable)

# Remarques

- Transmettre le retour de l'opérateur par référence permet d'écrire des instructions telles que `set['a'] = 12;`
  - Car c'est une lvalue
- Si l'on veut empêcher ce comportement, mais toujours éviter les copies, on peut retourner un `const`
- Souvent, les deux implémentations sont fournies
  - La variante `const` est appelée sur les objets `const`
- En général, la communauté préfère éviter de créer du code permettant d'utiliser `[] []`
  - Complexe
  - L'opérateur `[]` est unaire
  - Utiliser l'opérateur `()` (arité variable)

# Remarques

- Transmettre le retour de l'opérateur par référence permet d'écrire des instructions telles que `set['a'] = 12;`
  - Car c'est une lvalue
- Si l'on veut empêcher ce comportement, mais toujours éviter les copies, on peut retourner un `const`
- Souvent, les deux implémentations sont fournies
  - La variante `const` est appelée sur les objets `const`
- En général, la communauté préfère éviter de créer du code permettant d'utiliser `[] []`
  - Complexe
  - L'opérateur `[]` est unaire
  - Utiliser l'opérateur `()` (arité variable)

# Remarques

- Transmettre le retour de l'opérateur par référence permet d'écrire des instructions telles que `set['a'] = 12;`
  - Car c'est une lvalue
- Si l'on veut empêcher ce comportement, mais toujours éviter les copies, on peut retourner un `const`
- Souvent, les deux implémentations sont fournies
  - La variante `const` est appelée sur les objets `const`
- En général, la communauté préfère éviter de créer du code permettant d'utiliser `[] []`
  - Complexe
  - L'opérateur `[]` est unaire
  - Utiliser l'opérateur `()` (arité variable)

# Remarques

- Transmettre le retour de l'opérateur par référence permet d'écrire des instructions telles que `set['a'] = 12;`
  - Car c'est une lvalue
- Si l'on veut empêcher ce comportement, mais toujours éviter les copies, on peut retourner un `const`
- Souvent, les deux implémentations sont fournies
  - La variante `const` est appelée sur les objets `const`
- En général, la communauté préfère éviter de créer du code permettant d'utiliser `[] []`
  - Complexe
  - L'opérateur `[]` est unaire
  - Utiliser l'opérateur `()` (arité variable)



# Remarques

- Transmettre le retour de l'opérateur par référence permet d'écrire des instructions telles que `set['a'] = 12;`
  - Car c'est une lvalue
- Si l'on veut empêcher ce comportement, mais toujours éviter les copies, on peut retourner un `const`
- Souvent, les deux implémentations sont fournies
  - La variante `const` est appelée sur les objets `const`
- En général, la communauté préfère éviter de créer du code permettant d'utiliser `[] []`
  - Complexe
  - L'opérateur `[]` est unaire
  - Utiliser l'opérateur `()` (arité variable)

# Incrémentation et décrémentation

- Possibilité d'être utilisé en préfixé et en suffixé

## Deux prototypes

- Préfixé : `A& operator ++()`
- Suffixé : `A operator ++(int)` (paramètre ignoré)

- Très utile pour l'itération avec une boucle `foreach`
  - Classe itérable : définir `begin()` et `end`
  - Classe d'itérateur : définir `++` (préfixé), `!=` et `*` (indirection)
- Mêmes principes avec `-`

# Incrémentation et décrémentation

- Possibilité d'être utilisé en préfixé et en suffixé

## Deux prototypes

- 1 Préfixé : `A& operator ++()`
- 2 Suffixé : `A operator ++(int)` (paramètre ignoré)

- Très utile pour l'itération avec une boucle `foreach`
  - Classe itérable : définir `begin()` et `end`
  - Classe d'itérateur : définir `++` (préfixé), `!=` et `*` (indirection)
- Mêmes principes avec `-`

# Incrémentation et décrémentation

- Possibilité d'être utilisé en préfixé et en suffixé

## Deux prototypes

- 1 Préfixé : `A& operator ++()`
- 2 Suffixé : `A operator ++(int)` (paramètre ignoré)

- Très utile pour l'itération avec une boucle `foreach`
  - Classe itérable : définir `begin()` et `end`
  - Classe d'itérateur : définir `++` (préfixé), `!=` et `*` (indirection)
- Mêmes principes avec `-`

# Incrémentation et décrémentation

- Possibilité d'être utilisé en préfixé et en suffixé

## Deux prototypes

- 1 Préfixé : `A& operator ++()`
- 2 Suffixé : `A operator ++(int)` (paramètre ignoré)

- Très utile pour l'itération avec une boucle `foreach`
  - Classe itérable : définir `begin()` et `end`
  - Classe d'itérateur : définir `++` (préfixé), `!=` et `*` (indirection)
- Mêmes principes avec `-`

# Incrémentation et décrémentation

- Possibilité d'être utilisé en préfixé et en suffixé

## Deux prototypes

- 1 Préfixé : `A& operator ++()`
- 2 Suffixé : `A operator ++(int)` (paramètre ignoré)

- Très utile pour l'itération avec une boucle `foreach`
  - 1 Classe itérable : définir `begin()` et `end`
  - 2 Classe d'itérateur : définir `++` (préfixé), `!=` et `*` (indirection)
- Mêmes principes avec `-`

# Incrémentation et décrémentation

- Possibilité d'être utilisé en préfixé et en suffixé

## Deux prototypes

- 1 Préfixé : `A& operator ++()`
- 2 Suffixé : `A operator ++(int)` (paramètre ignoré)

- Très utile pour l'itération avec une boucle `foreach`
  - 1 Classe itérable : définir `begin()` et `end`
  - 2 Classe d'itérateur : définir `++` (préfixé), `!=` et `*` (indirection)
- Mêmes principes avec `-`

# Incrémentation et décrémentation

- Possibilité d'être utilisé en préfixé et en suffixé

## Deux prototypes

- 1 Préfixé : `A& operator ++()`
- 2 Suffixé : `A operator ++(int)` (paramètre ignoré)

- Très utile pour l'itération avec une boucle `foreach`
  - 1 Classe itérable : définir `begin()` et `end`
  - 2 Classe d'itérateur : définir `++` (préfixé), `!=` et `*` (indirection)
- Mêmes principes avec `-`



# Incrémentation et décrémentation

- Possibilité d'être utilisé en préfixé et en suffixé

## Deux prototypes

- 1 Préfixé : `A& operator ++()`
- 2 Suffixé : `A operator ++(int)` (paramètre ignoré)

- Très utile pour l'itération avec une boucle `foreach`
  - 1 Classe itérable : définir `begin()` et `end`
  - 2 Classe d'itérateur : définir `++` (préfixé), `!=` et `*` (indirection)
- Mêmes principes avec `-`

# Exemple

## ■ Fichier `increment.cpp`

```

1  class Integer
2  {
3      int i;
4      public:
5          Integer(int i = 0) : i(i) {}
6          friend ostream& operator <<(ostream&, const Integer&);
7
8          Integer& operator ++() { cout << "prefix" << endl; i++; return *this; } // prefix
9
10         Integer operator ++(int) // suffix
11         {
12             cout << "suffix" << endl;
13             Integer r = *this;
14             operator ++();
15             return r;
16         }
17     };
18
19     int main()
20     {
21         Integer i(2); Integer j = i;
22         cout << i++ << endl;
23         cout << ++j << endl;
24     }

```

# Les classes itérables

- On a vu qu'il est possible d'itérer sur certaines classes
  - `for(int i : v) ...`
- La surcharge d'opérateur permet à une classe d'utiliser cette syntaxe

## Grammaire pour une classe T itérable

- Déclarer dans T deux fonctions : `begin()` et `end()`
- La classe d'itérateur doit déclarer

# Les classes itérables

- On a vu qu'il est possible d'itérer sur certaines classes
  - `for(int i : v) ...`
- La surcharge d'opérateur permet à une classe d'utiliser cette syntaxe

## Grammaire pour une classe T itérable

- Déclarer dans T deux fonctions : `begin()` et `end()`
- La classe d'itérateur doit déclarer

# Les classes itérables

- On a vu qu'il est possible d'itérer sur certaines classes
  - `for(int i : v) ...`
- La surcharge d'opérateur permet à une classe d'utiliser cette syntaxe

## Grammaire pour une classe T itérable

- Déclarer dans T deux fonctions : `begin()` et `end()`
- La classe d'itérateur doit déclarer

# Les classes itérables

- On a vu qu'il est possible d'itérer sur certaines classes
  - `for (int i : v) ...`
- La surcharge d'opérateur permet à une classe d'utiliser cette syntaxe

## Grammaire pour une classe T itérable

- 1 Déclarer dans T deux fonctions : `begin()` et `end()`
  - Ces fonctions retournent un « itérateur »
- 2 La classe d'itérateur doit déclarer
  - un opérateur \* d'indirection
  - un opérateur < pour d'accéder à la donnée suivante
  - un opérateur ++ préfixé
  - un opérateur <== pour d'accéder à la donnée précédente
  - Un opérateur !=

# Les classes itérables

- On a vu qu'il est possible d'itérer sur certaines classes
  - `for (int i : v) ...`
- La surcharge d'opérateur permet à une classe d'utiliser cette syntaxe

## Grammaire pour une classe T itérable

### 1 Déclarer dans T deux fonctions : `begin()` et `end()`

- Ces fonctions retournent un « itérateur »

### 2 La classe d'itérateur doit déclarer

- un opérateur \* d'indirection
- un opérateur < d'ordre
- un opérateur ++ préfixé
- un opérateur <=
- Un opérateur !=

# Les classes itérables

- On a vu qu'il est possible d'itérer sur certaines classes
  - `for (int i : v) ...`
- La surcharge d'opérateur permet à une classe d'utiliser cette syntaxe

## Grammaire pour une classe T itérable

- 1 Déclarer dans T deux fonctions : `begin()` et `end()`
  - Ces fonctions retournent un « itérateur »
- 2 La classe d'itérateur doit déclarer
  - un opérateur \* d'indirection
  - un opérateur < d'ordre
  - un opérateur ++ préfixé
  - Un opérateur !=



# Les classes itérables

- On a vu qu'il est possible d'itérer sur certaines classes
  - `for (int i : v) ...`
- La surcharge d'opérateur permet à une classe d'utiliser cette syntaxe

## Grammaire pour une classe T itérable

- 1 Déclarer dans T deux fonctions : `begin()` et `end()`
  - Ces fonctions retournent un « itérateur »
- 2 La classe d'itérateur doit déclarer
  - un opérateur `*` d'indirection
    - Permet d'accéder à la donnée en cours
  - un opérateur `++` préfixé
    - Permet d'avancer l'itérateur
  - Un opérateur `!=`

# Les classes itérables

- On a vu qu'il est possible d'itérer sur certaines classes
  - `for (int i : v) ...`
- La surcharge d'opérateur permet à une classe d'utiliser cette syntaxe

## Grammaire pour une classe T itérable

- 1 Déclarer dans T deux fonctions : `begin()` et `end()`
  - Ces fonctions retournent un « itérateur »
- 2 La classe d'itérateur doit déclarer
  - un opérateur `*` d'indirection
    - Permet d'accéder à la donnée en cours
  - un opérateur `++` préfixé
    - Permet d'avancer l'itérateur
  - Un opérateur `!=`

# Les classes itérables

- On a vu qu'il est possible d'itérer sur certaines classes
  - `for (int i : v) ...`
- La surcharge d'opérateur permet à une classe d'utiliser cette syntaxe

## Grammaire pour une classe T itérable

- 1 Déclarer dans T deux fonctions : `begin()` et `end()`
  - Ces fonctions retournent un « itérateur »
- 2 La classe d'itérateur doit déclarer
  - un opérateur `*` d'indirection
    - Permet d'accéder à la donnée en cours
  - un opérateur `++` préfixé
    - Permet d'avancer l'itérateur
  - Un opérateur `!=`

# Les classes itérables

- On a vu qu'il est possible d'itérer sur certaines classes
  - `for (int i : v) ...`
- La surcharge d'opérateur permet à une classe d'utiliser cette syntaxe

## Grammaire pour une classe T itérable

- 1 Déclarer dans T deux fonctions : `begin()` et `end()`
  - Ces fonctions retournent un « itérateur »
- 2 La classe d'itérateur doit déclarer
  - un opérateur `*` d'indirection
    - Permet d'accéder à la donnée en cours
  - un opérateur `++` préfixé
    - Permet d'avancer l'itérateur
  - Un opérateur `!=`

# Les classes itérables

- On a vu qu'il est possible d'itérer sur certaines classes
  - `for (int i : v) ...`
- La surcharge d'opérateur permet à une classe d'utiliser cette syntaxe

## Grammaire pour une classe T itérable

- 1 Déclarer dans T deux fonctions : `begin()` et `end()`
  - Ces fonctions retournent un « itérateur »
- 2 La classe d'itérateur doit déclarer
  - un opérateur `*` d'indirection
    - Permet d'accéder à la donnée en cours
  - un opérateur `++` préfixé
    - Permet d'avancer l'itérateur
  - Un opérateur `!=`

# Les classes itérables

- On a vu qu'il est possible d'itérer sur certaines classes
  - `for (int i : v) ...`
- La surcharge d'opérateur permet à une classe d'utiliser cette syntaxe

## Grammaire pour une classe T itérable

- 1 Déclarer dans T deux fonctions : `begin()` et `end()`
  - Ces fonctions retournent un « itérateur »
- 2 La classe d'itérateur doit déclarer
  - un opérateur `*` d'indirection
    - Permet d'accéder à la donnée en cours
  - un opérateur `++` préfixé
    - Permet d'avancer l'itérateur
  - Un opérateur `!=`

# Exemple d'itération

## ■ Fichier `linkedlist.cpp`

```

1  class Nodelerator
2  {
3      Node* current;
4
5      public:
6          Nodelerator(Node * current) : current(current) {}
7
8          int operator *() { return current->data(); }
9
10         Nodelerator& operator ++() { current = current->next(); return *this; }
11
12         bool operator !=(const Nodelerator& it) const { return current != it.current; }
13     };
14
15     class LinkedList
16     {
17         Node* head; Node* tail;
18
19         public:
20             LinkedList() : head(nullptr), tail(nullptr) {}
21
22             Nodelerator begin() { return Nodelerator(head); }
23             Nodelerator end() { return Nodelerator(nullptr); }
24     };

```

# Surcharge de `new` et `delete`

- `new` et `delete` peuvent s'appliquer à des types de base ou des classes
- `new[]` et `delete[]` s'appliquent à des tableaux

## Surdéfinition

- Locale : pour une classe donnée. Les opérateurs « globaux » ont leur signification habituelle
- Globale : effectifs partout où une surdéfinition n'a pas été définie par l'utilisateur
- Redéfinir `new` et `delete` ne redéfinit pas `new[]` et `delete[]`
  - `new[]` et `delete[]` se surdéfinissent de la même façon



# Surcharge de `new` et `delete`

- `new` et `delete` peuvent s'appliquer à des types de base ou des classes
- `new []` et `delete []` s'appliquent à des tableaux

## Surdéfinition

- Locale : pour une classe donnée. Les opérateurs « globaux » ont leur signification habituelle
  - Globale : effectifs partout où une surdéfinition n'a pas été définie par l'utilisateur
- 
- Redéfinir `new` et `delete` ne redéfinit pas `new []` et `delete []`
    - `new []` et `delete []` se surdéfinissent de la même façon

# Surcharge de `new` et `delete`

- `new` et `delete` peuvent s'appliquer à des types de base ou des classes
- `new []` et `delete []` s'appliquent à des tableaux

## Surdéfinition

- 1 Locale : pour une classe donnée. Les opérateurs « globaux » ont leur signification habituelle
  - 2 Globale : effectifs partout où une surdéfinition n'a pas été définie par l'utilisateur
- Redéfinir `new` et `delete` ne redéfinit pas `new []` et `delete []`
    - `new []` et `delete []` se surdéfinissent de la même façon

# Surcharge de `new` et `delete`

- `new` et `delete` peuvent s'appliquer à des types de base ou des classes
- `new []` et `delete []` s'appliquent à des tableaux

## Surdéfinition

- 1 Locale : pour une classe donnée. Les opérateurs « globaux » ont leur signification habituelle
  - 2 Globale : effectifs partout où une surdéfinition n'a pas été définie par l'utilisateur
- Redéfinir `new` et `delete` ne redéfinit pas `new []` et `delete []`
    - `new []` et `delete []` se surdéfinissent de la même façon

# Surcharge de `new` et `delete`

- `new` et `delete` peuvent s'appliquer à des types de base ou des classes
- `new []` et `delete []` s'appliquent à des tableaux

## Surdéfinition

- 1 Locale : pour une classe donnée. Les opérateurs « globaux » ont leur signification habituelle
- 2 Globale : effectifs partout où une surdéfinition n'a pas été définie par l'utilisateur

- Redéfinir `new` et `delete` ne redéfinit pas `new []` et `delete []`
  - `new []` et `delete []` se surdéfinissent de la même façon

# Surcharge de `new` et `delete`

- `new` et `delete` peuvent s'appliquer à des types de base ou des classes
- `new []` et `delete []` s'appliquent à des tableaux

## Surdéfinition

- 1 Locale : pour une classe donnée. Les opérateurs « globaux » ont leur signification habituelle
  - 2 Globale : effectifs partout où une surdéfinition n'a pas été définie par l'utilisateur
- Redéfinir `new` et `delete` ne redéfinit pas `new []` et `delete []`
    - `new []` et `delete []` se surdéfinissent de la même façon

# Surcharge de `new` et `delete`

- `new` et `delete` peuvent s'appliquer à des types de base ou des classes
- `new []` et `delete []` s'appliquent à des tableaux

## Surdéfinition

- 1 Locale : pour une classe donnée. Les opérateurs « globaux » ont leur signification habituelle
  - 2 Globale : effectifs partout où une surdéfinition n'a pas été définie par l'utilisateur
- Redéfinir `new` et `delete` ne redéfinit pas `new []` et `delete []`
    - `new []` et `delete []` se surdéfinissent de la même façon

# Prototypes

## Surdéfinition de `new`

- Doit posséder un paramètre de type `size_t` (défini dans `cstdint.h`)
  - Correspond à la taille en octets de l'objet à allouer
  - Ne doit pas être spécifié à l'appel, le compilateur le gère seul
- Doit fournir en retour une valeur de type `void*` correspondant à l'adresse de l'objet alloué

## Surdéfinition de `delete`

- Doit recevoir un paramètre de type pointeur, fourni à l'appel
- Ne fournit aucun type de retour (`void`)

# Prototypes

## Surdéfinition de `new`

- Doit posséder un paramètre de type `size_t` (défini dans `cstdint.h`)
  - Correspond à la taille en octets de l'objet à allouer
  - Ne doit pas être spécifié à l'appel, le compilateur le gère seul
- Doit fournir en retour une valeur de type `void*` correspondant à l'adresse de l'objet alloué

## Surdéfinition de `delete`

- Doit recevoir un paramètre de type pointeur, fourni à l'appel
- Ne fournit aucun type de retour (`void`)



# Prototypes

## Surdéfinition de `new`

- Doit posséder un paramètre de type `size_t` (défini dans `cstdint.h`)
  - Correspond à la taille en octets de l'objet à allouer
    - Ne doit pas être spécifié à l'appel, le compilateur le gère seul
- Doit fournir en retour une valeur de type `void*` correspondant à l'adresse de l'objet alloué

## Surdéfinition de `delete`

- Doit recevoir un paramètre de type pointeur, fourni à l'appel
- Ne fournit aucun type de retour (`void`)

# Prototypes

## Surdéfinition de `new`

- Doit posséder un paramètre de type `size_t` (défini dans `cstdint.h`)
  - Correspond à la taille en octets de l'objet à allouer
  - Ne doit pas être spécifié à l'appel, le compilateur le gère seul
- Doit fournir en retour une valeur de type `void*` correspondant à l'adresse de l'objet alloué

## Surdéfinition de `delete`

- Doit recevoir un paramètre de type pointeur, fourni à l'appel
- Ne fournit aucun type de retour (`void`)

# Prototypes

## Surdéfinition de `new`

- Doit posséder un paramètre de type `size_t` (défini dans `cstdint.h`)
  - Correspond à la taille en octets de l'objet à allouer
  - Ne doit pas être spécifié à l'appel, le compilateur le gère seul
- Doit fournir en retour une valeur de type `void*` correspondant à l'adresse de l'objet alloué

## Surdéfinition de `delete`

- Doit recevoir un paramètre de type pointeur, fourni à l'appel
- Ne fournit aucun type de retour (`void`)

# Prototypes

## Surdéfinition de `new`

- Doit posséder un paramètre de type `size_t` (défini dans `cstdint.h`)
  - Correspond à la taille en octets de l'objet à allouer
  - Ne doit pas être spécifié à l'appel, le compilateur le gère seul
- Doit fournir en retour une valeur de type `void*` correspondant à l'adresse de l'objet alloué

## Surdéfinition de `delete`

- Doit recevoir un paramètre de type pointeur, fourni à l'appel
  - Représente l'adresse de l'emplacement alloué à libérer
- Ne fournit aucun type de retour (`void`)

# Prototypes

## Surdéfinition de `new`

- Doit posséder un paramètre de type `size_t` (défini dans `cstdint.h`)
  - Correspond à la taille en octets de l'objet à allouer
  - Ne doit pas être spécifié à l'appel, le compilateur le gère seul
- Doit fournir en retour une valeur de type `void*` correspondant à l'adresse de l'objet alloué

## Surdéfinition de `delete`

- Doit recevoir un paramètre de type pointeur, fourni à l'appel
  - Représente l'adresse de l'emplacement alloué à libérer
- Ne fournit aucun type de retour (`void`)

# Prototypes

## Surdéfinition de `new`

- Doit posséder un paramètre de type `size_t` (défini dans `csdefs.h`)
  - Correspond à la taille en octets de l'objet à allouer
  - Ne doit pas être spécifié à l'appel, le compilateur le gère seul
- Doit fournir en retour une valeur de type `void*` correspondant à l'adresse de l'objet alloué

## Surdéfinition de `delete`

- Doit recevoir un paramètre de type pointeur, fourni à l'appel
  - Représente l'adresse de l'emplacement alloué à libérer
- Ne fournit aucun type de retour (`void`)

# Prototypes

## Surdéfinition de `new`

- Doit posséder un paramètre de type `size_t` (défini dans `cstdint.h`)
  - Correspond à la taille en octets de l'objet à allouer
  - Ne doit pas être spécifié à l'appel, le compilateur le gère seul
- Doit fournir en retour une valeur de type `void*` correspondant à l'adresse de l'objet alloué

## Surdéfinition de `delete`

- Doit recevoir un paramètre de type pointeur, fourni à l'appel
  - Représente l'adresse de l'emplacement alloué à libérer
- Ne fournit aucun type de retour (`void`)

# Remarques

- Les surdéfinitions n'ont d'incidence que pour les objets dynamiques
- Que `new` soit surdéfini ou non, son appel est systématiquement suivi d'un appel constructeur
- Que `delete` soit surdéfini ou non, son appel est systématiquement suivi d'un appel destructeur
- Pour une définition
  - globale, il faut utiliser une fonction indépendante
  - locale, il faut utiliser une fonction membre
- Il est possible, lors de la surcharge de `new` et `delete`, de faire appel aux opérateurs originaux via `::`
- `new` et `delete` sont statiques
  - Elles n'ont accès qu'aux membres statiques
  - Elles ne sont pas appelées via le paramètre implicite `this`



# Remarques

- Les surdéfinitions n'ont d'incidence que pour les objets dynamiques
- Que `new` soit surdéfini ou non, son appel est systématiquement suivi d'un appel constructeur
- Que `delete` soit surdéfini ou non, son appel est systématiquement suivi d'un appel destructeur
- Pour une définition
  - globale, il faut utiliser une fonction indépendante
  - locale, il faut utiliser une fonction membre
- Il est possible, lors de la surcharge de `new` et `delete`, de faire appel aux opérateurs originaux via `::`
- `new` et `delete` sont statiques
  - Elles n'ont accès qu'aux membres statiques
  - Elles ne sont pas appelées via le paramètre implicite `this`

# Remarques

- Les surdéfinitions n'ont d'incidence que pour les objets dynamiques
- Que `new` soit surdéfini ou non, son appel est systématiquement suivi d'un appel constructeur
- Que `delete` soit surdéfini ou non, son appel est systématiquement suivi d'un appel destructeur
- Pour une définition
  - globale, il faut utiliser une fonction indépendante
  - locale, il faut utiliser une fonction membre
- Il est possible, lors de la surcharge de `new` et `delete`, de faire appel aux opérateurs originaux via `::`
- `new` et `delete` sont statiques
  - Elles n'ont accès qu'aux membres statiques
  - Elles ne sont pas appelées via le paramètre implicite `this`

# Remarques

- Les surdéfinitions n'ont d'incidence que pour les objets dynamiques
- Que `new` soit surdéfini ou non, son appel est systématiquement suivi d'un appel constructeur
- Que `delete` soit surdéfini ou non, son appel est systématiquement suivi d'un appel destructeur
- Pour une définition
  - 1 globale, il faut utiliser une fonction indépendante
  - 2 locale, il faut utiliser une fonction membre
- Il est possible, lors de la surcharge de `new` et `delete`, de faire appel aux opérateurs originaux via `::`
- `new` et `delete` sont statiques
  - Elles n'ont accès qu'aux membres statiques
  - Elles ne sont pas appelées via le paramètre implicite `this`

# Remarques

- Les surdéfinitions n'ont d'incidence que pour les objets dynamiques
- Que `new` soit surdéfini ou non, son appel est systématiquement suivi d'un appel constructeur
- Que `delete` soit surdéfini ou non, son appel est systématiquement suivi d'un appel destructeur
- Pour une définition
  - 1 globale, il faut utiliser une fonction indépendante
  - 2 locale, il faut utiliser une fonction membre
- Il est possible, lors de la surcharge de `new` et `delete`, de faire appel aux opérateurs originaux via `::`
- `new` et `delete` sont statiques
  - Elles n'ont accès qu'aux membres statiques
  - Elles ne sont pas appelées via le paramètre implicite `this`

# Remarques

- Les surdéfinitions n'ont d'incidence que pour les objets dynamiques
- Que `new` soit surdéfini ou non, son appel est systématiquement suivi d'un appel constructeur
- Que `delete` soit surdéfini ou non, son appel est systématiquement suivi d'un appel destructeur
- Pour une définition
  - 1 globale, il faut utiliser une fonction indépendante
  - 2 locale, il faut utiliser une fonction membre
- Il est possible, lors de la surcharge de `new` et `delete`, de faire appel aux opérateurs originaux via `::`
- `new` et `delete` sont statiques
  - Elles n'ont accès qu'aux membres statiques
  - Elles ne sont pas appelées via le paramètre implicite `this`

# Remarques

- Les surdéfinitions n'ont d'incidence que pour les objets dynamiques
- Que `new` soit surdéfini ou non, son appel est systématiquement suivi d'un appel constructeur
- Que `delete` soit surdéfini ou non, son appel est systématiquement suivi d'un appel destructeur
- Pour une définition
  - 1 globale, il faut utiliser une fonction indépendante
  - 2 locale, il faut utiliser une fonction membre
- Il est possible, lors de la surcharge de `new` et `delete`, de faire appel aux opérateurs originaux via : :
- `new` et `delete` sont statiques
  - Elles n'ont accès qu'aux membres statiques
  - Elles ne sont pas appelées via le paramètre implicite `this`

# Remarques

- Les surdéfinitions n'ont d'incidence que pour les objets dynamiques
- Que `new` soit surdéfini ou non, son appel est systématiquement suivi d'un appel constructeur
- Que `delete` soit surdéfini ou non, son appel est systématiquement suivi d'un appel destructeur
- Pour une définition
  - 1 globale, il faut utiliser une fonction indépendante
  - 2 locale, il faut utiliser une fonction membre
- Il est possible, lors de la surcharge de `new` et `delete`, de faire appel aux opérateurs originaux via `::`
- `new` et `delete` sont statiques
  - Elles n'ont accès qu'aux membres statiques
  - Elles ne sont pas appelées via le paramètre implicite `this`

# Remarques

- Les surdéfinitions n'ont d'incidence que pour les objets dynamiques
- Que `new` soit surdéfini ou non, son appel est systématiquement suivi d'un appel constructeur
- Que `delete` soit surdéfini ou non, son appel est systématiquement suivi d'un appel destructeur
- Pour une définition
  - 1 globale, il faut utiliser une fonction indépendante
  - 2 locale, il faut utiliser une fonction membre
- Il est possible, lors de la surcharge de `new` et `delete`, de faire appel aux opérateurs originaux via `::`
- `new` et `delete` sont statiques
  - Elles n'ont accès qu'aux membres statiques
  - Elles ne sont pas appelées via le paramètre implicite `this`



# Remarques

- Les surdéfinitions n'ont d'incidence que pour les objets dynamiques
- Que `new` soit surdéfini ou non, son appel est systématiquement suivi d'un appel constructeur
- Que `delete` soit surdéfini ou non, son appel est systématiquement suivi d'un appel destructeur
- Pour une définition
  - 1 globale, il faut utiliser une fonction indépendante
  - 2 locale, il faut utiliser une fonction membre
- Il est possible, lors de la surcharge de `new` et `delete`, de faire appel aux opérateurs originaux via `::`
- `new` et `delete` sont statiques
  - Elles n'ont accès qu'aux membres statiques
  - Elles ne sont pas appelées via le paramètre implicite `this`

# Exemple

## ■ Fichier newdel.cpp

```

1  class point
2  {
3      static int n; static int nd;
4      int x, y;
5
6      public:
7          point(int abs=0, int ord=0) : x(abs), y(ord)
8          { n++; cout << "(+)_Number_of_points_:" << n << endl; }
9
10         ~point() { n--; cout << "(-)_Number_of_points_:" << n << endl; }
11
12         void * operator new(size_t size)
13         {
14             nd++; cout << "(+)_Number_of_dynamic_points_:" << nd << endl;
15             return ::new char[size];
16         }
17
18         void operator delete(void * pt)
19         { nd--; cout << "(-)_Number_of_dynamic_points_:" << nd << endl; }
20     };
21
22     int point::n = 0; //talk about that stuff
23     int point::nd = 0;

```

# Objets fonctions

# Fonctions en paramètres

- En C++, il est possible de passer des fonctions en paramètres d'autres fonctions
- Utile pour
  - appliquer une fonction à tous les objets d'un conteneur
  - filtrer des données (compter si  $x > 0$ )
  - fournir une fonction à exécuter au sein d'un thread, etc.
- Trois moyens de mise en œuvre
  - Les fonctions indépendantes
  - Les lambdas
  - les objets fonctions (foncteurs)

## Remarque

- Impossible de passer une fonction *inline* en paramètre

# Fonctions en paramètres

- En C++, il est possible de passer des fonctions en paramètres d'autres fonctions
- Utile pour
  - appliquer une fonction à tous les objets d'un conteneur
  - filtrer des données (compter si  $x > 0$ )
  - fournir une fonction à exécuter au sein d'un thread, etc.
- Trois moyens de mise en œuvre
  - Les fonctions indépendantes
  - Les lambdas
  - les objets fonctions (foncteurs)

## Remarque

- Impossible de passer une fonction *inline* en paramètre

# Fonctions en paramètres

- En C++, il est possible de passer des fonctions en paramètres d'autres fonctions
- Utile pour
  - appliquer une fonction à tous les objets d'un conteneur
  - filtrer des données (compter si  $x > 0$ )
  - fournir une fonction à exécuter au sein d'un thread, etc.
- Trois moyens de mise en œuvre
  - Les fonctions indépendantes
  - Les lambdas
  - les objets fonctions (foncteurs)

## Remarque

- Impossible de passer une fonction *en ligne* en paramètre

# Fonctions en paramètres

- En C++, il est possible de passer des fonctions en paramètres d'autres fonctions
- Utile pour
  - appliquer une fonction à tous les objets d'un conteneur
  - filtrer des données (compter si  $x > 0$ )
  - fournir une fonction à exécuter au sein d'un thread, etc.
- Trois moyens de mise en œuvre
  - Les fonctions indépendantes
  - Les lambdas
  - les objets fonctions (foncteurs)

## Remarque

- Impossible de passer une fonction *en ligne* en paramètre

# Fonctions en paramètres

- En C++, il est possible de passer des fonctions en paramètres d'autres fonctions
- Utile pour
  - appliquer une fonction à tous les objets d'un conteneur
  - filtrer des données (compter si  $x > 0$ )
  - fournir une fonction à exécuter au sein d'un thread, etc.
- Trois moyens de mise en œuvre
  - Les fonctions indépendantes
  - Les lambdas
  - les objets fonctions (foncteurs)

## Remarque

- Impossible de passer une fonction *en ligne* en paramètre



# Fonctions en paramètres

- En C++, il est possible de passer des fonctions en paramètres d'autres fonctions
- Utile pour
  - appliquer une fonction à tous les objets d'un conteneur
  - filtrer des données (compter si  $x > 0$ )
  - fournir une fonction à exécuter au sein d'un thread, etc.
- Trois moyens de mise en œuvre
  - 1 Les fonctions indépendantes
  - 2 Les lambdas
  - 3 les objets fonctions (foncteurs)

## Remarque

- Impossible de passer une fonction en ligne en paramètre

# Fonctions en paramètres

- En C++, il est possible de passer des fonctions en paramètres d'autres fonctions
- Utile pour
  - appliquer une fonction à tous les objets d'un conteneur
  - filtrer des données (compter si  $x > 0$ )
  - fournir une fonction à exécuter au sein d'un thread, etc.
- Trois moyens de mise en œuvre
  - 1 Les fonctions indépendantes
  - 2 Les lambdas
  - 3 les objets fonctions (foncteurs)

## Remarque

- Impossible de passer une fonction *en ligne* en paramètre

# Fonctions en paramètres

- En C++, il est possible de passer des fonctions en paramètres d'autres fonctions
- Utile pour
  - appliquer une fonction à tous les objets d'un conteneur
  - filtrer des données (compter si  $x > 0$ )
  - fournir une fonction à exécuter au sein d'un thread, etc.
- Trois moyens de mise en œuvre
  - 1 Les fonctions indépendantes
  - 2 Les lambdas
  - 3 les objets fonctions (foncteurs)

## Remarque

- Impossible de passer une fonction en ligne en paramètre

# Fonctions en paramètres

- En C++, il est possible de passer des fonctions en paramètres d'autres fonctions
- Utile pour
  - appliquer une fonction à tous les objets d'un conteneur
  - filtrer des données (compter si  $x > 0$ )
  - fournir une fonction à exécuter au sein d'un thread, etc.
- Trois moyens de mise en œuvre
  - 1 Les fonctions indépendantes
  - 2 Les lambdas
  - 3 les objets fonctions (foncteurs)

## Remarque

- Impossible de passer une fonction en ligne en paramètre

# Fonctions en paramètres

- En C++, il est possible de passer des fonctions en paramètres d'autres fonctions
- Utile pour
  - appliquer une fonction à tous les objets d'un conteneur
  - filtrer des données (compter si  $x > 0$ )
  - fournir une fonction à exécuter au sein d'un thread, etc.
- Trois moyens de mise en œuvre
  - 1 Les fonctions indépendantes
  - 2 Les lambdas
  - 3 les objets fonctions (foncteurs)

## Remarque

- Impossible de passer une fonction `inline` en paramètre

# Fonctions en paramètres

- En C++, il est possible de passer des fonctions en paramètres d'autres fonctions
- Utile pour
  - appliquer une fonction à tous les objets d'un conteneur
  - filtrer des données (compter si  $x > 0$ )
  - fournir une fonction à exécuter au sein d'un thread, etc.
- Trois moyens de mise en œuvre
  - 1 Les fonctions indépendantes
  - 2 Les lambdas
  - 3 les objets fonctions (foncteurs)

## Remarque

- Impossible de passer une fonction `inline` en paramètre

# Objets fonctions

## ■ Mis en œuvre via la surcharge d'opérateur

■ B `operator()` (A a)

- La fonction membre `()` prend en paramètre un A et retourne un B

## Constructeur par défaut

- Si l'on veut passer un tel objet en paramètre « comme une fonction », il doit posséder un constructeur par défaut
- Sinon, il faut le créer au préalable

# Objets fonctions

- Mis en œuvre via la surcharge d'opérateur
- B `operator()` (A a)
  - La fonction membre `()` prend en paramètre un A et retourne un B

## Constructeur par défaut

- Si l'on veut passer un tel objet en paramètre « comme une fonction », il doit posséder un constructeur par défaut
- Sinon, il faut le créer au préalable



# Objets fonctions

- Mis en œuvre via la surcharge d'opérateur
- B `operator()` (A a)
  - La fonction membre `()` prend en paramètre un A et retourne un B

## Constructeur par défaut

- Si l'on veut passer un tel objet en paramètre « comme une fonction », il doit posséder un constructeur par défaut
- Sinon, il faut le créer au préalable

# Objets fonctions

- Mis en œuvre via la surcharge d'opérateur
- `B operator() (A a)`
  - La fonction membre `()` prend en paramètre un `A` et retourne un `B`

## Constructeur par défaut

- Si l'on veut passer un tel objet en paramètre « comme une fonction », il doit posséder un constructeur par défaut
- Sinon, il faut le créer au préalable

# Objets fonctions

- Mis en œuvre via la surcharge d'opérateur
- `B operator() (A a)`
  - La fonction membre `()` prend en paramètre un `A` et retourne un `B`

## Constructeur par défaut

- Si l'on veut passer un tel objet en paramètre « comme une fonction », il doit posséder un constructeur par défaut
- Sinon, il faut le créer au préalable

# Objets fonctions

- Mis en œuvre via la surcharge d'opérateur
- `B operator() (A a)`
  - La fonction membre `()` prend en paramètre un `A` et retourne un `B`

## Constructeur par défaut

- Si l'on veut passer un tel objet en paramètre « comme une fonction », il doit posséder un constructeur par défaut
- Sinon, il faut le créer au préalable

# Exemple (1/2)

## ■ Fichier `foncteur.cpp`

```
1  class Tada //try to remove default cstr
2  {
3      public:
4          void operator () (int n)
5          {
6              cout << "Tada_" << n << endl;
7          }
8  };
9
10 void f(int& n)
11 {
12     cout << "Applying_f_on_" << n << endl;
13     n = n * 2;
14     if (n % 3 == 0)
15         n++;
16 }
17
18 bool impair(int n)
19 {
20     return n % 2 == 1;
21 }
```

## Exemple (2/2)

### ■ Fichier `foncteur.cpp`

```
1  int main()
2  {
3      vector<int> v = {1, 2, 3, 4, 5, 6};
4
5      for_each(v.begin(), v.end(), f);
6      cout << endl;
7
8      for_each(v.begin(), v.end(), Tada()); //try to build Tada before
9      cout << endl;
10
11     auto result = find_if(v.begin(), v.end(), impair);
12     while(result != v.end())
13     {
14         cout << *result << endl;
15         result++;
16     }
17 }
```

# Les comparateurs

- La librairie standard permet d'effectuer un traitement sur des objets comparables
  - Trier
  - Trouver le maximum
  - Remplir un deque
- Pour qu'un deux objets soient comparables, il faut qu'ils définissent un opérateur <
  - On peut le surcharger si nécessaire
  - Fonction membre ou indépendante
- On peut également fournir un « comparateur »
  - Avec une fonction lambda
  - En surchargeant l'opérateur `bool (T, T)`

# Les comparateurs

- La librairie standard permet d'effectuer un traitement sur des objets comparables
  - Trier
    - Trouver le maximum
    - Remplir un deque
- Pour qu'un deux objets soient comparables, il faut qu'ils définissent un opérateur <
  - On peut le surcharger si nécessaire
  - Fonction membre ou indépendante
- On peut également fournir un « comparateur »
  - Avec une fonction lambda
  - En surchargeant l'opérateur `bool (T, T)`



# Les comparateurs

- La librairie standard permet d'effectuer un traitement sur des objets comparables
  - Trier
  - Trouver le maximum
  - Remplir un deque
- Pour qu'un deux objets soient comparables, il faut qu'ils définissent un opérateur <
  - On peut le surcharger si nécessaire
  - Fonction membre ou indépendante
- On peut également fournir un « comparateur »
  - Avec une fonction lambda
  - En surchargeant l'opérateur `bool (T, T)`

# Les comparateurs

- La librairie standard permet d'effectuer un traitement sur des objets comparables
  - Trier
  - Trouver le maximum
  - Remplir un `deque`
- Pour qu'un deux objets soient comparables, il faut qu'ils définissent un opérateur `<`
  - On peut le surcharger si nécessaire
  - Fonction membre ou indépendante
- On peut également fournir un « comparateur »
  - Avec une fonction lambda
  - En surchargeant l'opérateur `bool (T, T)`

# Les comparateurs

- La librairie standard permet d'effectuer un traitement sur des objets comparables
  - Trier
  - Trouver le maximum
  - Remplir un `deque`
- Pour qu'un deux objets soient comparables, il faut qu'ils définissent un opérateur `<`
  - On peut le surcharger si nécessaire
  - Fonction membre ou indépendante
- On peut également fournir un « comparateur »
  - Avec une fonction lambda
  - En surchargeant l'opérateur `bool (T, T)`

# Les comparateurs

- La librairie standard permet d'effectuer un traitement sur des objets comparables
  - Trier
  - Trouver le maximum
  - Remplir un `deque`
- Pour qu'un deux objets soient comparables, il faut qu'ils définissent un opérateur `<`
  - On peut le surcharger si nécessaire
    - Fonction membre ou indépendante
- On peut également fournir un « comparateur »
  - Avec une fonction lambda
  - En surchargeant l'opérateur `bool (T, T)`

# Les comparateurs

- La librairie standard permet d'effectuer un traitement sur des objets comparables
  - Trier
  - Trouver le maximum
  - Remplir un `deque`
- Pour qu'un deux objets soient comparables, il faut qu'ils définissent un opérateur `<`
  - On peut le surcharger si nécessaire
  - Fonction membre ou indépendante
- On peut également fournir un « comparateur »
  - Avec une fonction lambda
  - En surchargeant l'opérateur `bool (T, T)`

# Les comparateurs

- La librairie standard permet d'effectuer un traitement sur des objets comparables
  - Trier
  - Trouver le maximum
  - Remplir un `deque`
- Pour qu'un deux objets soient comparables, il faut qu'ils définissent un opérateur `<`
  - On peut le surcharger si nécessaire
  - Fonction membre ou indépendante
- On peut également fournir un « comparateur »
  - Avec une fonction lambda
  - En surchargeant l'opérateur `bool (T, T)`

# Les comparateurs

- La librairie standard permet d'effectuer un traitement sur des objets comparables
  - Trier
  - Trouver le maximum
  - Remplir un `deque`
- Pour qu'un deux objets soient comparables, il faut qu'ils définissent un opérateur `<`
  - On peut le surcharger si nécessaire
  - Fonction membre ou indépendante
- On peut également fournir un « comparateur »
  - Avec une fonction lambda
  - En surchargeant l'opérateur `bool (T, T)`

# Les comparateurs

- La librairie standard permet d'effectuer un traitement sur des objets comparables
  - Trier
  - Trouver le maximum
  - Remplir un `deque`
- Pour qu'un deux objets soient comparables, il faut qu'ils définissent un opérateur `<`
  - On peut le surcharger si nécessaire
  - Fonction membre ou indépendante
- On peut également fournir un « comparateur »
  - Avec une fonction lambda
  - En surchargeant l'opérateur `bool (T, T)`



# Exemple : opérateur <

## ■ Fichier comparator.cpp

```

1  class IntegerOp
2  {
3      int i;
4      public:
5          IntegerOp(int i) : i(i) {}
6
7          int& value() { return i; }
8          const int& value() const { return i; }
9
10         bool operator<(const IntegerOp& other) const
11         {
12             return i < other.i;
13         }
14     }
15
16     int main()
17     {
18         vector<IntegerOp> v = {IntegerOp(3), IntegerOp(5), IntegerOp(2),
19                               IntegerOp(1), IntegerOp(4)};
20         sort(v.begin(), v.end()); //ok : IntegerOp has <
21         for(IntegerOp i : v)
22             cout << i << " ";
23     }

```

# Exemple : lambda et foncteur

## ■ Fichier comparator.cpp

```

1  class Integer {
2      ... // no bool operator <(Integer)
3  };
4
5  int main() {
6      vector<Integer> v1 = {Integer(3), Integer(5), Integer(2), Integer(1), Integer(4)};
7      vector<Integer> v2 = v1;
8      // sort(v1.begin(), v1.end()); //ko : no < in Integer
9      sort(v1.begin(), v1.end(), [](const Integer& i1, const Integer& i2) {
10         return i1.value() < i2.value(); }); //lambda
11     for(Integer i : v1)
12         cout << i << " ";
13     cout << endl;
14
15     struct IntegerComparator {
16         bool operator()(const Integer& i1, const Integer& i2) const {
17             return i1.value() < i2.value();
18         }
19     };
20
21     sort(v2.begin(), v2.end(), IntegerComparator()); //function object
22     for(Integer i : v2)
23         cout << i << " ";
24     cout << endl;
25 }

```

# Exemple avec les conteneurs

## ■ Fichier `comparator.cpp`

```

1  int main() {
2      //priority_queue<Integer> p; //ko : no < in Integer
3      priority_queue<Integer, deque<Integer>, IntegerComparator> p;
4      p.push(Integer(3));
5      p.push(Integer(5));
6      p.push(Integer(2));
7      p.push(Integer(1));
8      p.push(Integer(4));
9      while(! p.empty()) {
10         cout << p.top() << "␣";
11         p.pop();
12     }
13     cout << endl;
14
15     //    key    value    key_comparator
16     map<Integer, int, IntegerComparator> m;
17     m[Integer(3)] = 3;
18     m[Integer(5)] = 5;
19     m[Integer(2)] = 2;
20     m[Integer(1)] = 1;
21     m[Integer(4)] = 4;
22     for(auto p : m)
23         cout << p.second << "␣";
24     cout << endl;
25 }
```

# Allocations dynamiques

# Code suspect

- Considérez la classe `vector` suivante
- Fichier `vector-bad.cpp`

```
1  class vector
2  {
3      int n;
4      double * tab;
5
6      public:
7          vector(int nbr) : n(nbr), tab(new double[n]) {}
8
9          ~vector()
10         {
11             delete[] tab;
12         }
13
14         double & operator [] (int i)
15         {
16             return tab[i];
17         }
18     };
```

# Problème

## Détection attaque sournoise 3D20

■ TU AS FAIT UN NEW !

Question n° 1 : que fait le code suivant ?

```
1 vector v(5);
```

```
2 f(v);
```

Question n° 2 : que fait le code suivant ?

```
1 vector v1(5); vector v2(6)
```

```
2 v2 = v1;
```

# Problème

## Détection attaque sournoise 3D20

■ TU AS FAIT UN NEW !

Question n° 1 : que fait le code suivant ?

```
1 vector v(5);
```

```
2 f(v);
```

Question n° 2 : que fait le code suivant ?

```
1 vector v1(5); vector v2(6)
```

```
2 v2 = v1;
```

# Problème

## Détection attaque sournoise 3D20

■ TU AS FAIT UN NEW !

Question n° 1 : que fait le code suivant ?

```
1 vector v(5);  
2 f(v);
```

Question n° 2 : que fait le code suivant ?

```
1 vector v1(5); vector v2(6)  
2 v2 = v1;
```



# Problème

## Détection attaque sournoise 3D20

■ TU AS FAIT UN NEW !

Question n° 1 : que fait le code suivant ?

```
1 vector v(5);
```

```
2 f(v);
```

Question n° 2 : que fait le code suivant ?

```
1 vector v1(5); vector v2(6)
```

```
2 v2 = v1;
```

# Problème

## Détection attaque sournoise 3D20

■ TU AS FAIT UN NEW !

Question n° 1 : que fait le code suivant ?

```
1 vector v(5);  
2 f(v);
```

Question n° 2 : que fait le code suivant ?

```
1 vector v1(5); vector v2(6)  
2 v2 = v1;
```

# Problème

## Détection attaque sournoise 3D20

■ TU AS FAIT UN NEW !

Question n° 1 : que fait le code suivant ?

```
1 vector v(5);  
2 f(v);
```

Question n° 2 : que fait le code suivant ?

```
1 vector v1(5); vector v2(6)  
2 v2 = v1;
```

# Problème

## Détection attaque sournoise 3D20

■ TU AS FAIT UN `NEW` !

Question n° 1 : que fait le code suivant ?

```
1 vector v(5);  
2 f(v);
```

Question n° 2 : que fait le code suivant ?

```
1 vector v1(5); vector v2(6)  
2 v2 = v1;
```

# Problème

## Détection attaque sournoise 3D20

■ TU AS FAIT UN `NEW` !

Question n° 1 : que fait le code suivant ?

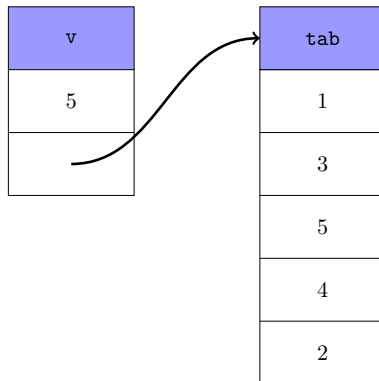
```
1 vector v(5);  
2 f(v);
```

Question n° 2 : que fait le code suivant ?

```
1 vector v1(5); vector v2(6)  
2 v2 = v1;
```

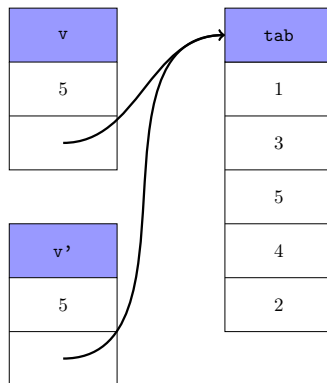
## Question n° 1 : appel de $f(v)$ (1/3)

### ■ Instanciation du vecteur $v$



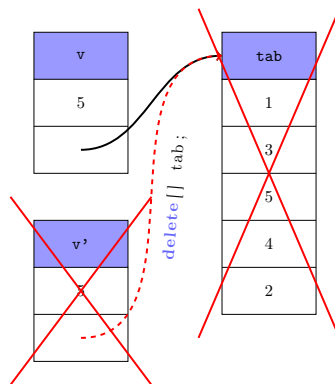
## Question n° 1 : appel de $f(v)$ (2/3)

### ■ Création de copie locale $v'$ à l'appel



## Question n° 1 : appel de $f(v)$ (3/3)

- Destruction de copie locale  $v'$  en sortie de  $f$





# Problèmes

## Observation

- 1 À l'appel, le vecteur  $v$  est copié, ainsi que l'adresse du tableau
  - Pas la mémoire allouée sur le tas
- 2 Quand la copie est détruite, le destructeur libère la mémoire

## Problème

- Le vecteur  $v$  est dans un état incohérent après l'appel
- On ne veut peut-être pas utiliser `shared_ptr`
  - Effet de bord sur les copies
- Deux solutions possibles
  - Copie « manuelle »
  - Empêcher la copie

# Problèmes

## Observation

- 1 À l'appel, le vecteur  $v$  est copié, ainsi que l'adresse du tableau
  - Pas la mémoire allouée sur le tas
- 2 Quand la copie est détruite, le destructeur libère la mémoire

## Problème

- Le vecteur  $v$  est dans un état incohérent après l'appel
- On ne veut peut-être pas utiliser `shared_ptr`
  - Effet de bord sur les copies
- Deux solutions possibles
  - Copie « manuelle »
  - Empêcher la copie

# Problèmes

## Observation

- 1 À l'appel, le vecteur  $v$  est copié, ainsi que l'adresse du tableau
  - Pas la mémoire allouée sur le tas
- 2 Quand la copie est détruite, le destructeur libère la mémoire

## Problème

- Le vecteur  $v$  est dans un état incohérent après l'appel
- On ne veut peut-être pas utiliser `shared_ptr`
  - Effet de bord sur les copies
- Deux solutions possibles
  - Copie « manuelle »
  - Empêcher la copie

# Problèmes

## Observation

- 1 À l'appel, le vecteur  $v$  est copié, ainsi que l'adresse du tableau
  - Pas la mémoire allouée sur le tas
- 2 Quand la copie est détruite, le destructeur libère la mémoire

## Problème

- Le vecteur  $v$  est dans un état incohérent après l'appel
- On ne veut peut-être pas utiliser `shared_ptr`
  - Effet de bord sur les copies
- Deux solutions possibles
  - Copie « manuelle »
  - Empêcher la copie

# Problèmes

## Observation

- 1 À l'appel, le vecteur  $v$  est copié, ainsi que l'adresse du tableau
  - Pas la mémoire allouée sur le tas
- 2 Quand la copie est détruite, le destructeur libère la mémoire

## Problème

- Le vecteur  $v$  est dans un état incohérent après l'appel
- On ne veut peut-être pas utiliser `shared_ptr`
  - Effet de bord sur les copies
- Deux solutions possibles
  - Copie « manuelle »
  - Empêcher la copie

# Problèmes

## Observation

- 1 À l'appel, le vecteur  $v$  est copié, ainsi que l'adresse du tableau
  - Pas la mémoire allouée sur le tas
- 2 Quand la copie est détruite, le destructeur libère la mémoire

## Problème

- Le vecteur  $v$  est dans un état incohérent après l'appel
- On ne veut peut-être pas utiliser `shared_ptr`
  - Effet de bord sur les copies
- Deux solutions possibles
  - Copie « manuelle »
  - Empêcher la copie

# Problèmes

## Observation

- 1 À l'appel, le vecteur  $v$  est copié, ainsi que l'adresse du tableau
  - Pas la mémoire allouée sur le tas
- 2 Quand la copie est détruite, le destructeur libère la mémoire

## Problème

- Le vecteur  $v$  est dans un état incohérent après l'appel
- On ne veut peut-être pas utiliser `shared_ptr`
  - Effet de bord sur les copies
- Deux solutions possibles
  - Copie « manuelle »
  - Empêcher la copie

# Problèmes

## Observation

- 1 À l'appel, le vecteur  $v$  est copié, ainsi que l'adresse du tableau
  - Pas la mémoire allouée sur le tas
- 2 Quand la copie est détruite, le destructeur libère la mémoire

## Problème

- Le vecteur  $v$  est dans un état incohérent après l'appel
- On ne veut peut-être pas utiliser `shared_ptr`
  - Effet de bord sur les copies
- Deux solutions possibles
  - Copie « manuelle »
  - Empêcher la copie



# Problèmes

## Observation

- 1 À l'appel, le vecteur  $v$  est copié, ainsi que l'adresse du tableau
  - Pas la mémoire allouée sur le tas
- 2 Quand la copie est détruite, le destructeur libère la mémoire

## Problème

- Le vecteur  $v$  est dans un état incohérent après l'appel
- On ne veut peut-être pas utiliser `shared_ptr`
  - Effet de bord sur les copies
- Deux solutions possibles
  - 1 Copie « manuelle »
  - 2 Empêcher la copie

# Problèmes

## Observation

- 1 À l'appel, le vecteur  $v$  est copié, ainsi que l'adresse du tableau
  - Pas la mémoire allouée sur le tas
- 2 Quand la copie est détruite, le destructeur libère la mémoire

## Problème

- Le vecteur  $v$  est dans un état incohérent après l'appel
- On ne veut peut-être pas utiliser `shared_ptr`
  - Effet de bord sur les copies
- Deux solutions possibles
  - 1 Copie « manuelle »
  - 2 Empêcher la copie

# Problèmes

## Observation

- 1 À l'appel, le vecteur  $v$  est copié, ainsi que l'adresse du tableau
  - Pas la mémoire allouée sur le tas
- 2 Quand la copie est détruite, le destructeur libère la mémoire

## Problème

- Le vecteur  $v$  est dans un état incohérent après l'appel
- On ne veut peut-être pas utiliser `shared_ptr`
  - Effet de bord sur les copies
- Deux solutions possibles
  - 1 Copie « manuelle »
  - 2 Empêcher la copie

# Copie « manuelle »

- Mise en œuvre via un constructeur de recopie
- Écrire un constructeur de recopie qui copie manuellement les données allouées sur le tas

## Avantages

- Copie fonctionnelle

## Inconvénient

- Temps
- Mémoire

# Copie « manuelle »

- Mise en œuvre via un constructeur de recopie
- Écrire un constructeur de recopie qui copie manuellement les données allouées sur le tas

## Avantages

- Copie fonctionnelle

## Inconvénient

- Temps
- Mémoire

# Copie « manuelle »

- Mise en œuvre via un constructeur de recopie
- Écrire un constructeur de recopie qui copie manuellement les données allouées sur le tas

## Avantages

- Copie fonctionnelle

## Inconvénient

- Temps
- Mémoire

# Copie « manuelle »

- Mise en œuvre via un constructeur de recopie
- Écrire un constructeur de recopie qui copie manuellement les données allouées sur le tas

## Avantages

- Copie fonctionnelle

## Inconvénient

- Temps
- Mémoire

# Copie « manuelle »

- Mise en œuvre via un constructeur de recopie
- Écrire un constructeur de recopie qui copie manuellement les données allouées sur le tas

## Avantages

- Copie fonctionnelle

## Inconvénient

- Temps
- Mémoire



# Copie « manuelle »

- Mise en œuvre via un constructeur de recopie
- Écrire un constructeur de recopie qui copie manuellement les données allouées sur le tas

## Avantages

- Copie fonctionnelle

## Inconvénient

- Temps
- Mémoire

# Copie « manuelle »

- Mise en œuvre via un constructeur de recopie
- Écrire un constructeur de recopie qui copie manuellement les données allouées sur le tas

## Avantages

- Copie fonctionnelle

## Inconvénient

- Temps
- Mémoire

■ Fichier `vector-copy.cpp`

```

1  class vector
2  {
3      int n;
4      double * tab;
5
6      public:
7          vector(int nbr) : n(nbr), tab(new double[n]) {}
8
9          vector(const vector & v)
10         {
11             n = v.n;
12             tab = new double[n];
13             for(int i = 0; i < n; i++)
14                 tab[i] = v.tab[i];
15         }
16
17         ~vector()
18         {
19             delete[] tab;
20         }
21
22         double & operator [] (int i)
23         {
24             return tab[i];
25         }
26 };

```

# Copie interdite

- Mise en œuvre via un constructeur déclaré `delete` (C++11)
  - Pré C++11 : constructeur de recopie privé, ou déclaré mais pas implémenté

## Avantages

- Rapide

## Inconvénient

- Peut-être pas ce qu'on veut

# Copie interdite

- Mise en œuvre via un constructeur déclaré `delete` (C++11)
  - Pré C++11 : constructeur de recopie privé, ou déclaré mais pas implémenté

## Avantages

- Rapide

## Inconvénient

- Peut-être pas ce qu'on veut

# Copie interdite

- Mise en œuvre via un constructeur déclaré `delete` (C++11)
  - Pré C++11 : constructeur de recopie privé, ou déclaré mais pas implémenté

## Avantages

- Rapide

## Inconvénient

- Peut-être pas ce qu'on veut

# Copie interdite

- Mise en œuvre via un constructeur déclaré `delete` (C++11)
  - Pré C++11 : constructeur de recopie privé, ou déclaré mais pas implémenté

## Avantages

- Rapide

## Inconvénient

- Peut-être pas ce qu'on veut

# Copie interdite

- Mise en œuvre via un constructeur déclaré `delete` (C++11)
  - Pré C++11 : constructeur de recopie privé, ou déclaré mais pas implémenté

## Avantages

- Rapide

## Inconvénient

- Peut-être pas ce qu'on veut



# Copie interdite

- Mise en œuvre via un constructeur déclaré `delete` (C++11)
  - Pré C++11 : constructeur de recopie privé, ou déclaré mais pas implémenté

## Avantages

- Rapide

## Inconvénient

- Peut-être pas ce qu'on veut

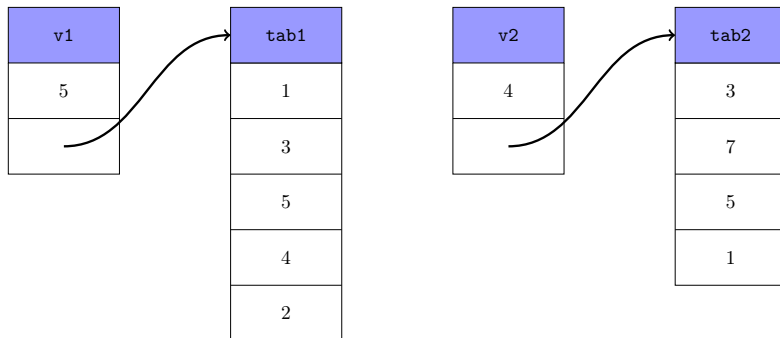
## ■ Fichier `vector-del.cpp`

```
1  class vector
2  {
3      int n;
4      double * tab;
5
6      public:
7          vector(int nbr) : n(nbr), tab(new double[n]) {}
8
9          vector(const vector & v) = delete;
10
11         ~vector()
12         {
13             delete[] tab;
14         }
15
16         double & operator [] (int i)
17         {
18             return tab[i];
19         }
20     };
```

## ■ Utiliser le passage par référence lors des appels de fonctions

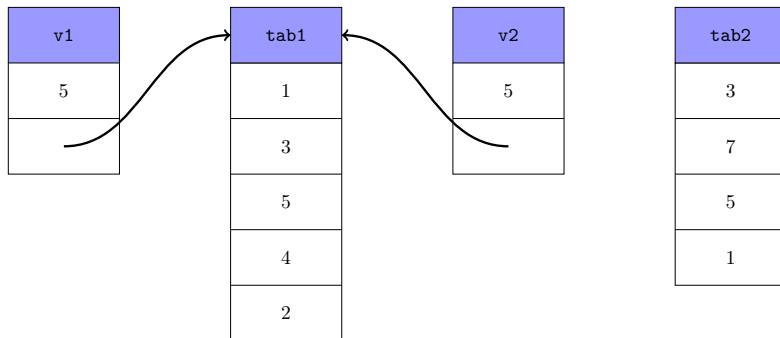
## Question n° 2 : affectation de $v1$ à $v2$ (1/3)

### ■ Instanciation des vecteurs $v1$ et $v2$



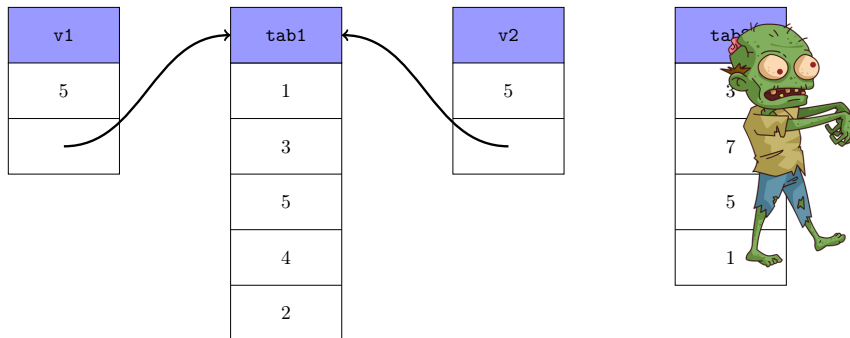
## Question n° 2 : affectation de $v1$ à $v2$ (2/3)

### ■ Affectation de $v1$ à $v2$



## Question n° 2 : affectation de $v1$ à $v2$ (3/3)

- Fuite mémoire : le tableau de  $v2$  n'a pas été détruit



# Problèmes

## Observation

- 1 À l'affectation, le vecteur `v2` est affecté
  - Les variables automatiques sont cohérentes
- 2 Quand la copie `v1` *ou* `v2` sont détruits, `tab1` est libéré
  - Pas `tab2`

## Problème

- Fuite mémoire : `tab2` n'est pas désalloué
- Problème potentiel : double delete

- Deux solutions possibles
  - Affectation « manuelle » de « recopie »
  - Empêcher l'affectation

# Problèmes

## Observation

- 1 À l'affectation, le vecteur `v2` est affecté
  - Les variables automatiques sont cohérentes
- 2 Quand la copie `v1` ou `v2` sont détruits, `tab1` est libéré
  - Pas `tab2`

## Problème

- Fuite mémoire : `tab2` n'est pas désalloué
- Problème potentiel : double delete
- Deux solutions possibles
  - Affectation « manuelle » de « recopie »
  - Empêcher l'affectation

# Problèmes

## Observation

- 1 À l'affectation, le vecteur `v2` est affecté
  - Les variables automatiques sont cohérentes
- 2 Quand la copie `v1` ou `v2` sont détruits, `tab1` est libéré
  - Pas `tab2`

## Problème

- Fuite mémoire : `tab2` n'est pas désalloué
- Problème potentiel : double delete

- Deux solutions possibles
  - Affectation « manuelle » de « recopie »
  - Empêcher l'affectation



# Problèmes

## Observation

- 1 À l'affectation, le vecteur `v2` est affecté
  - Les variables automatiques sont cohérentes
- 2 Quand la copie `v1` *ou* `v2` sont détruits, `tab1` est libéré
  - Pas `tab2`

## Problème

- Fuite mémoire : `tab2` n'est pas désalloué
- Problème potentiel : double delete

- Deux solutions possibles
  - Affectation « manuelle » de « recopie »
  - Empêcher l'affectation

# Problèmes

## Observation

- 1 À l'affectation, le vecteur `v2` est affecté
  - Les variables automatiques sont cohérentes
- 2 Quand la copie `v1` *ou* `v2` sont détruits, `tab1` est libéré
  - Pas `tab2`

## Problème

- Fuite mémoire : `tab2` n'est pas désalloué
- Problème potentiel : double delete

- Deux solutions possibles
  - Affectation « manuelle » de « recopie »
  - Empêcher l'affectation

# Problèmes

## Observation

- 1 À l'affectation, le vecteur `v2` est affecté
  - Les variables automatiques sont cohérentes
- 2 Quand la copie `v1` *ou* `v2` sont détruits, `tab1` est libéré
  - Pas `tab2`

## Problème

- Fuite mémoire : `tab2` n'est pas désalloué
- Problème potentiel : double `delete`

- Deux solutions possibles
  - Affectation « manuelle » de « recopie »
  - Empêcher l'affectation

# Problèmes

## Observation

- 1 À l'affectation, le vecteur `v2` est affecté
  - Les variables automatiques sont cohérentes
- 2 Quand la copie `v1` *ou* `v2` sont détruits, `tab1` est libéré
  - Pas `tab2`

## Problème

- Fuite mémoire : `tab2` n'est pas désalloué
- Problème potentiel : double *delete*

- Deux solutions possibles
  - Affectation « manuelle » de « recopie »
  - Empêcher l'affectation

# Problèmes

## Observation

- 1 À l'affectation, le vecteur `v2` est affecté
  - Les variables automatiques sont cohérentes
- 2 Quand la copie `v1` *ou* `v2` sont détruits, `tab1` est libéré
  - Pas `tab2`

## Problème

- Fuite mémoire : `tab2` n'est pas désalloué
- Problème potentiel : double `delete`

- Deux solutions possibles

- Affectation « manuelle » de « recopie »
- Empêcher l'affectation

# Problèmes

## Observation

- 1 À l'affectation, le vecteur `v2` est affecté
  - Les variables automatiques sont cohérentes
- 2 Quand la copie `v1` *ou* `v2` sont détruits, `tab1` est libéré
  - Pas `tab2`

## Problème

- Fuite mémoire : `tab2` n'est pas désalloué
- Problème potentiel : double `delete`

- Deux solutions possibles

- 1 Affectation « manuelle » de « recopie »
- 2 Empêcher l'affectation

# Problèmes

## Observation

- 1 À l'affectation, le vecteur `v2` est affecté
  - Les variables automatiques sont cohérentes
- 2 Quand la copie `v1` *ou* `v2` sont détruits, `tab1` est libéré
  - Pas `tab2`

## Problème

- Fuite mémoire : `tab2` n'est pas désalloué
- Problème potentiel : double `delete`

- Deux solutions possibles

- 1 Affectation « manuelle » de « recopie »
- 2 Empêcher l'affectation

# Problèmes

## Observation

- 1 À l'affectation, le vecteur `v2` est affecté
  - Les variables automatiques sont cohérentes
- 2 Quand la copie `v1` *ou* `v2` sont détruits, `tab1` est libéré
  - Pas `tab2`

## Problème

- Fuite mémoire : `tab2` n'est pas désalloué
- Problème potentiel : double `delete`

- Deux solutions possibles

- 1 Affectation « manuelle » de « recopie »
- 2 Empêcher l'affectation



# Affectation « manuelle »

- Mise en œuvre via surcharge de =
- Écrire un opérateur = qui affecte et détruit manuellement les données allouées sur le tas

## Avantages

- Affectation fonctionnelle

## Inconvénient

- Temps
- Mémoire
- Complexité si ce n'est pas une recopie

# Affectation « manuelle »

- Mise en œuvre via surcharge de =
- Écrire un opérateur = qui affecte et détruit manuellement les données allouées sur le tas

## Avantages

- Affectation fonctionnelle

## Inconvénient

- Temps
- Mémoire
- Complexité si ce n'est pas une recopie

## Affectation « manuelle »

- Mise en œuvre via surcharge de =
- Écrire un opérateur = qui affecte et détruit manuellement les données allouées sur le tas

### Avantages

- Affectation fonctionnelle

### Inconvénient

- Temps
- Mémoire
- Complexité si ce n'est pas une copie

## Affectation « manuelle »

- Mise en œuvre via surcharge de =
- Écrire un opérateur = qui affecte et détruit manuellement les données allouées sur le tas

### Avantages

- Affectation fonctionnelle

### Inconvénient

- Temps
- Mémoire
- Complexité si ce n'est pas une recopie

## Affectation « manuelle »

- Mise en œuvre via surcharge de =
- Écrire un opérateur = qui affecte et détruit manuellement les données allouées sur le tas

### Avantages

- Affectation fonctionnelle

### Inconvénient

- Temps
- Mémoire
- Complexité si ce n'est pas une recopie

## Affectation « manuelle »

- Mise en œuvre via surcharge de =
- Écrire un opérateur = qui affecte et détruit manuellement les données allouées sur le tas

### Avantages

- Affectation fonctionnelle

### Inconvénient

- Temps
- Mémoire
- Complexité si ce n'est pas une recopie

## Affectation « manuelle »

- Mise en œuvre via surcharge de =
- Écrire un opérateur = qui affecte et détruit manuellement les données allouées sur le tas

### Avantages

- Affectation fonctionnelle

### Inconvénient

- Temps
- Mémoire
- Complexité si ce n'est pas une recopie

# Affectation « manuelle »

- Mise en œuvre via surcharge de =
- Écrire un opérateur = qui affecte et détruit manuellement les données allouées sur le tas

## Avantages

- Affectation fonctionnelle

## Inconvénient

- Temps
- Mémoire
- Complexité si ce n'est pas une recopie



■ Fichier `vector-copy.cpp`

```

1  class vector
2  {
3      int n;
4      int count_affect;
5      double * tab;
6
7      public:
8          vector(int nbr) : n(nbr), tab(new double[n]), count_affect(0) {}
9
10         ~vector() { delete[] tab; }
11
12         vector& operator = (const vector& v)
13         {
14             if(this != &v) //check self-assign
15             {
16                 delete tab;
17
18                 n = v.n;
19                 tab = new double[n];
20                 for(int i = 0; i < n; i++)
21                     tab[i] = v.tab[i];
22             }
23             return * this;
24         }
25
26         double & operator [] (int i)
27         {
28             return tab[i];
29         }
30 };

```

# Remarques

- Dans le cas de surcharge de `=`, comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite affecter un vecteur constant à un vecteur quelconque
- On retourne le résultat de l'affectation par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie
- Si on ne veut pas recopier les données à l'affectation
  - Création d'un attribut compteur d'affectations
  - Le destructeur détruit `tab` si ce compteur est à zéro
  - L'opérateur détruit `tab` si le compteur de `v` est à zéro

## Hygiène de programmation

- Ne faites pas de `new`

# Remarques

- Dans le cas de surcharge de `=`, comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite affecter un vecteur constant à un vecteur quelconque
- On retourne le résultat de l'affectation par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie
- Si on ne veut pas recopier les données à l'affectation
  - Création d'un attribut compteur d'affectations
  - Le destructeur détruit `tab` si ce compteur est à zéro
  - L'opérateur détruit `tab` si le compteur de `v` est à zéro

## Hygiène de programmation

- Ne faites pas de `new`

# Remarques

- Dans le cas de surcharge de `=`, comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite affecter un vecteur constant à un vecteur quelconque
- On retourne le résultat de l'affectation par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie
- Si on ne veut pas recopier les données à l'affectation
  - Création d'un attribut compteur d'affectations
  - Le destructeur détruit `tab` si ce compteur est à zéro
  - L'opérateur détruit `tab` si le compteur de `v` est à zéro

## Hygiène de programmation

- Ne faites pas de `new`

# Remarques

- Dans le cas de surcharge de `=`, comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite affecter un vecteur constant à un vecteur quelconque
- On retourne le résultat de l'affectation par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de recopie
- Si on ne veut pas recopier les données à l'affectation
  - Création d'un attribut compteur d'affectations
  - Le destructeur détruit `tab` si ce compteur est à zéro
  - L'opérateur détruit `tab` si le compteur de `v` est à zéro

## Hygiène de programmation

- Ne faites pas de `new`

# Remarques

- Dans le cas de surcharge de `=`, comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite affecter un vecteur constant à un vecteur quelconque
- On retourne le résultat de l'affectation par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie
- Si on ne veut pas recopier les données à l'affectation
  - 1 Création d'un attribut compteur d'affectations
  - 2 Le destructeur détruit `tab` si ce compteur est à zéro
  - 3 L'opérateur détruit `tab` si le compteur de `v` est à zéro

## Hygiène de programmation

- Ne faites pas de `new`

# Remarques

- Dans le cas de surcharge de `=`, comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite affecter un vecteur constant à un vecteur quelconque
- On retourne le résultat de l'affectation par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie
- Si on ne veut pas recopier les données à l'affectation
  - 1 Création d'un attribut compteur d'affectations
  - 2 Le destructeur détruit `tab` si ce compteur est à zéro
  - 3 L'opérateur détruit `tab` si le compteur de `v` est à zéro

## Hygiène de programmation

- Ne faites pas de `new`

# Remarques

- Dans le cas de surcharge de `=`, comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite affecter un vecteur constant à un vecteur quelconque
- On retourne le résultat de l'affectation par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie
- Si on ne veut pas recopier les données à l'affectation
  - 1 Création d'un attribut compteur d'affectations
  - 2 Le destructeur détruit `tab` si ce compteur est à zéro
  - 3 L'opérateur détruit `tab` si le compteur de `v` est à zéro

## Hygiène de programmation

- Ne faites pas de `new`



# Remarques

- Dans le cas de surcharge de `=`, comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite affecter un vecteur constant à un vecteur quelconque
- On retourne le résultat de l'affectation par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie
- Si on ne veut pas recopier les données à l'affectation
  - 1 Création d'un attribut compteur d'affectations
  - 2 Le destructeur détruit `tab` si ce compteur est à zéro
  - 3 L'opérateur détruit `tab` si le compteur de `v` est à zéro

## Hygiène de programmation

- Ne faites pas de `new`

## Remarques

- Dans le cas de surcharge de `=`, comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite affecter un vecteur constant à un vecteur quelconque
- On retourne le résultat de l'affectation par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie
- Si on ne veut pas recopier les données à l'affectation
  - 1 Création d'un attribut compteur d'affectations
  - 2 Le destructeur détruit `tab` si ce compteur est à zéro
  - 3 L'opérateur détruit `tab` si le compteur de `v` est à zéro

## Hygiène de programmation

- Ne faites pas de `new`

## Remarques

- Dans le cas de surcharge de `=`, comme on transmet le paramètre par référence, on doit le déclarer `const` si on souhaite affecter un vecteur constant à un vecteur quelconque
- On retourne le résultat de l'affectation par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de copie
- Si on ne veut pas recopier les données à l'affectation
  - 1 Création d'un attribut compteur d'affectations
  - 2 Le destructeur détruit `tab` si ce compteur est à zéro
  - 3 L'opérateur détruit `tab` si le compteur de `v` est à zéro

## Hygiène de programmation

- Ne faites pas de `new`

# Affectation interdite

- Mise en œuvre via un opérateur = déclaré `delete` (C++11)
  - Pré C++11 : opérateur d'affectation privé, ou déclaré mais pas implémenté

## Avantages

- Rapide

## Inconvénient

- Peut-être pas ce qu'on veut

# Affectation interdite

- Mise en œuvre via un opérateur = déclaré `delete` (C++11)
  - Pré C++11 : opérateur d'affectation privé, ou déclaré mais pas implémenté

## Avantages

- Rapide

## Inconvénient

- Peut-être pas ce qu'on veut

# Affectation interdite

- Mise en œuvre via un opérateur = déclaré `delete` (C++11)
  - Pré C++11 : opérateur d'affectation privé, ou déclaré mais pas implémenté

## Avantages

- Rapide

## Inconvénient

- Peut-être pas ce qu'on veut

# Affectation interdite

- Mise en œuvre via un opérateur = déclaré `delete` (C++11)
  - Pré C++11 : opérateur d'affectation privé, ou déclaré mais pas implémenté

## Avantages

- Rapide

## Inconvénient

- Peut-être pas ce qu'on veut

# Affectation interdite

- Mise en œuvre via un opérateur = déclaré `delete` (C++11)
  - Pré C++11 : opérateur d'affectation privé, ou déclaré mais pas implémenté

## Avantages

- Rapide

## Inconvénient

- Peut-être pas ce qu'on veut



# Affectation interdite

- Mise en œuvre via un opérateur = déclaré `delete` (C++11)
  - Pré C++11 : opérateur d'affectation privé, ou déclaré mais pas implémenté

## Avantages

- Rapide

## Inconvénient

- Peut-être pas ce qu'on veut

## ■ Fichier `vector-del.cpp`

```

1  class vector
2  {
3      int n;
4      double * tab;
5
6      public:
7          vector(int nbr) : n(nbr), tab(new double[n]) {}
8
9          ~vector()
10         {
11             delete[] tab;
12         }
13
14         vector& operator = (const vector& v) = delete;
15
16         double & operator [] (int i)
17         {
18             return tab[i];
19         }
20 };

```

## ■ Construction possible, réaffectation impossible

- `vector v(3, brol); //ok`
- `vector v = vector(3, brol); //ko`
- `v1 = v2; //ko`

# Littéraux définis par l'utilisateur

# Introduction

- Supposons que l'on veuille modéliser le MRUA d'un point
- À l'évidence, le code ci-dessous ne compile pas

```

1  class MRUA
2  {
3      double x0, t0, v0, a;
4
5      public :
6          MRUA(double x0 = 0, double t0 = 0, double v0 = 0, double a = 9.81)
7              : x0 ( x0 ) , t0 ( t0 ) , v0 ( v0 ) , a ( a )
8              {}
9
10         double operator () (double t_in_seconds) { ... }
11         double operator () (double t_in_milliseconds) { ... }
12     };

```

- Il existe plusieurs techniques « standard » pour résoudre ce type d'ambiguïté
  - Typage fort, tag dispatch, types fantômes, etc.
  - Cf. Ch. 14
- En C++, dans ce cas précis, définir des littéraux dédiés est approprié

# Illustration

## ■ Fichier `mrua.cpp`

```

1  class MRUA {
2      long double x0, t0, v0, a;
3
4  public :
5      MRUA(long double x0 = 0, long double t0 = 0, long double v0 = 0, long double a = 9.81)
6          : x0 ( x0 ) , t0 ( t0 ) , v0 ( v0 ) , a ( a )
7          {}
8
9      long double operator () (long double t) {
10         long double dt = (t - t0);
11         return x0 + v0 * dt + 0.5 * dt * dt;
12     }
13 };
14
15 inline long double operator "" _ms(long double x) { return 0.001 * x; }
16 inline long double operator "" _s(long double x) { return x; }
17 inline long double operator "" _min(long double x) { return 60 * x; }
18 inline long double operator "" _h(long double x) { return 3600 * x; }
19
20 int main() {
21     MRUA mrua;
22     cout << "Position_after_1_millisecond_is_" << mrua(1.0_ms) << "_meters" << endl;
23     cout << "Position_after_1_second_is_" << mrua(1.0_s) << "_meters" << endl;
24     cout << "Position_after_1_minute_is_" << mrua(1.0_min) << "_meters" << endl;
25     cout << "Position_after_1_hour_is_" << mrua(1.0_h) << "_meters" << endl;
26 }

```

# Contraintes

- 1 Les littéraux définis par l'utilisateur sont introduits par la surcharge de l'opérateur `" "`
- 2 Le nom du littéral commence par `'_'`
  - La librairie standard se réserve les littéraux non préfixés
- 3 Les paramètres ne peuvent être que de type `const char*`, `unsigned long long`, `long double`, `char`, `« const char*, size_t »`
- 4 Les conversions implicites ne sont pas acceptées
  - `mrva(1_ms)` ne compile pas
- 5 Pas de valeur par défaut

## Littéraux standards existants

- Dans `chrono.h`, les littéraux de durée `ns`, `us`, `ms`, `s`, `min`, `h` sont définis
- Dans `complex.h`, les littéraux imaginaires `if`, `il`, `i` sont définis

# Contraintes

- 1 Les littéraux définis par l'utilisateur sont introduits par la surcharge de l'opérateur `" "`
- 2 Le nom du littéral commence par `'_'`
  - La librairie standard se réserve les littéraux non préfixés
- 3 Les paramètres ne peuvent être que de type `const char*`, `unsigned long long`, `long double`, `char`, `« const char*, size_t »`
- 4 Les conversions implicites ne sont pas acceptées
  - `mrva(1_ms)` ne compile pas
- 5 Pas de valeur par défaut

## Littéraux standards existants

- Dans `chrono.h`, les littéraux de durée `ns`, `us`, `ms`, `s`, `min`, `h` sont définis
- Dans `complex.h`, les littéraux imaginaires `if`, `il`, `i` sont définis

# Contraintes

- 1 Les littéraux définis par l'utilisateur sont introduits par la surcharge de l'opérateur `" "`
- 2 Le nom du littéral commence par `'_'`
  - La librairie standard se réserve les littéraux non préfixés
- 3 Les paramètres ne peuvent être que de type `const char*`, `unsigned long long`, `long double`, `char`, `« const char*, size_t »`
- 4 Les conversions implicites ne sont pas acceptées
  - `mrva(1_ms)` ne compile pas
- 5 Pas de valeur par défaut

## Littéraux standards existants

- Dans `chrono.h`, les littéraux de durée `ns`, `us`, `ms`, `s`, `min`, `h` sont définis
- Dans `complex.h`, les littéraux imaginaires `if`, `il`, `i` sont définis



# Contraintes

- 1 Les littéraux définis par l'utilisateur sont introduits par la surcharge de l'opérateur `" "`
- 2 Le nom du littéral commence par `'_'`
  - La librairie standard se réserve les littéraux non préfixés
- 3 Les paramètres ne peuvent être que de type `const char*`, `unsigned long long`, `long double`, `char`, `« const char*, size_t »`
- 4 Les conversions implicites ne sont pas acceptées
  - `mrva(1_ms)` ne compile pas
- 5 Pas de valeur par défaut

## Littéraux standards existants

- Dans `chrono.h`, les littéraux de durée `ns`, `us`, `ms`, `s`, `min`, `h` sont définis
- Dans `complex.h`, les littéraux imaginaires `if`, `il`, `i` sont définis

# Contraintes

- 1 Les littéraux définis par l'utilisateur sont introduits par la surcharge de l'opérateur `" "`
- 2 Le nom du littéral commence par `'_'`
  - La librairie standard se réserve les littéraux non préfixés
- 3 Les paramètres ne peuvent être que de type `const char*`, `unsigned long long`, `long double`, `char`, `« const char*, size_t »`
- 4 Les conversions implicites ne sont pas acceptées
  - `mrua(1_ms)` ne compile pas
- 5 Pas de valeur par défaut

## Littéraux standards existants

- Dans `chrono.h`, les littéraux de durée `ns`, `us`, `ms`, `s`, `min`, `h` sont définis
- Dans `complex.h`, les littéraux imaginaires `if`, `il`, `i` sont définis

# Contraintes

- 1 Les littéraux définis par l'utilisateur sont introduits par la surcharge de l'opérateur `" "`
- 2 Le nom du littéral commence par `'_'`
  - La librairie standard se réserve les littéraux non préfixés
- 3 Les paramètres ne peuvent être que de type `const char*`, `unsigned long long`, `long double`, `char`, `« const char*, size_t »`
- 4 Les conversions implicites ne sont pas acceptées
  - `mrua(1_ms)` ne compile pas
- 5 Pas de valeur par défaut

## Littéraux standards existants

- Dans `chrono.h`, les littéraux de durée `ns`, `us`, `ms`, `s`, `min`, `h` sont définis
- Dans `complex.h`, les littéraux imaginaires `if`, `il`, `i` sont définis

# Contraintes

- 1 Les littéraux définis par l'utilisateur sont introduits par la surcharge de l'opérateur `" "`
- 2 Le nom du littéral commence par `'_'`
  - La librairie standard se réserve les littéraux non préfixés
- 3 Les paramètres ne peuvent être que de type `const char*`, `unsigned long long`, `long double`, `char`, `« const char*, size_t »`
- 4 Les conversions implicites ne sont pas acceptées
  - `mrua(1_ms)` ne compile pas
- 5 Pas de valeur par défaut

## Littéraux standards existants

- Dans `chrono.h`, les littéraux de durée `ns`, `us`, `ms`, `s`, `min`, `h` sont définis
- Dans `complex.h`, les littéraux imaginaires `if`, `il`, `i` sont définis

# Contraintes

- 1 Les littéraux définis par l'utilisateur sont introduits par la surcharge de l'opérateur `" "`
- 2 Le nom du littéral commence par `'_'`
  - La librairie standard se réserve les littéraux non préfixés
- 3 Les paramètres ne peuvent être que de type `const char*`, `unsigned long long`, `long double`, `char`, `« const char*, size_t »`
- 4 Les conversions implicites ne sont pas acceptées
  - `mrua(1_ms)` ne compile pas
- 5 Pas de valeur par défaut

## Littéraux standards existants

- Dans `chrono.h`, les littéraux de durée `ns`, `us`, `ms`, `s`, `min`, `h` sont définis
- Dans `complex.h`, les littéraux imaginaires `if`, `il`, `i` sont définis

# Contraintes

- 1 Les littéraux définis par l'utilisateur sont introduits par la surcharge de l'opérateur `" "`
- 2 Le nom du littéral commence par `'_'`
  - La librairie standard se réserve les littéraux non préfixés
- 3 Les paramètres ne peuvent être que de type `const char*`, `unsigned long long`, `long double`, `char`,  
« `const char*`, `size_t` »
- 4 Les conversions implicites ne sont pas acceptées
  - `mrua(1_ms)` ne compile pas
- 5 Pas de valeur par défaut

## Littéraux standards existants

- Dans `chrono.h`, les littéraux de durée `ns`, `us`, `ms`, `s`, `min`, `h` sont définis
- Dans `complex.h`, les littéraux imaginaires `if`, `il`, `i` sont définis

# Contraintes

- 1 Les littéraux définis par l'utilisateur sont introduits par la surcharge de l'opérateur `" "`
- 2 Le nom du littéral commence par `'_'`
  - La librairie standard se réserve les littéraux non préfixés
- 3 Les paramètres ne peuvent être que de type `const char*`, `unsigned long long`, `long double`, `char`, `« const char*, size_t »`
- 4 Les conversions implicites ne sont pas acceptées
  - `mrua(1_ms)` ne compile pas
- 5 Pas de valeur par défaut

## Littéraux standards existants

- Dans `chrono.h`, les littéraux de durée `ns`, `us`, `ms`, `s`, `min`, `h` sont définis
- Dans `complex.h`, les littéraux imaginaires `if`, `il`, `i` sont définis