5 - Surcharge d'opérateurs

UE 15 - Informatique appliquée

F. Pluquet

HelHa
Slides originaux de R. Absil (ESI)

11 novembre 2018

Table des matières

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 (2/2)

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

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 (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

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 $\star$ préfixé sur un itérateur retourne la donnée pointée

Surcharge d'opérateur

Syntaxe

- Utilisation du mot-clé `operator`
- Deux utilisations 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

Exemple avec opérateur * 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));
14 }
```

Exemple avec opérateur * 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));
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 recopie, 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
    - Utile si gros objets
  - transmettre le retour par référence
  - allouer dynamiquement le retour (peu recommandé)
    - Overhead si petits objets
    - Gestion mémoire « complexe »
  - protéger les arguments contre la copie 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 `sstream` et `string`
    - Inefficace

En pratique

- Surcharge de « `(ostream&, const MaClasse& brol)`
- Avec une fonction amie, indépendante

- Comme on transmet le paramètre par référence, on doit le déclarer `const`
- On retourne le résultat de l'impression par référence afin
  - de pouvoir traiter les affectations multiples
  - d'éviter d'appeler le constructeur de recopie (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
  - `a + 2`
  - mais pas `2 + a`
- En indépendant, cette possibilité existe
  - Fournir deux implémentations

- Éviter la redondance (+, +=, symétrie, etc.)
  - Définir une implémentation
  - Les autres sont `inline` et appellent la première

Surcharge de `[]`

- Fichier `tab.cpp`

```
1  class Tab {
2
3    static const int LENGTH = 30;
4    char tabs[LENGTH];
5
6    public:
7      char& operator [](int index){
8        return tabs[index];
9      }
10
11     int size() const{
12       return LENGTH;
13     }
14 };
15
16 int main(){
17   Tab tab;
18   tab[4] = 'c';
19   for(int i = 0; i < tab.size(); i++) {
20     cout << "tab␣[" << i << "]" << tab[i] << endl;
21   }
22   return 0;
23 }
```

Remarques

- Transmettre le retour de l'opérateur par référence permet d'écrire des instructions telles que `tab[4] = 12;`
- Si l'on veut empêcher ce comportement, mais toujours éviter les copies, on peut retourner un `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

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 }
```

Exemple d'itération

- Fichier `linkedlist.cpp`

```
1  class NodeIterator
2  {
3    Node* current;
4
5    public:
6      NodeIterator(Node * current) : current(current) {}
7
8      int operator *() { return current->data(); }
9
10     NodeIterator& operator ++() { current = current->next(); return *this; }
11
12     bool operator !=(const NodeIterator& 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     NodeIterator begin() { return NodeIterator(head); }
23     NodeIterator end() { return NodeIterator(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

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 `cstsdef.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
  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 et lambdas

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 objets fonctions (foncteurs)
  3. Les lambdas

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

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 }
14
15 bool impair(int n)
16 {
17   return n % 2 == 1;
18 }
```

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 }
```

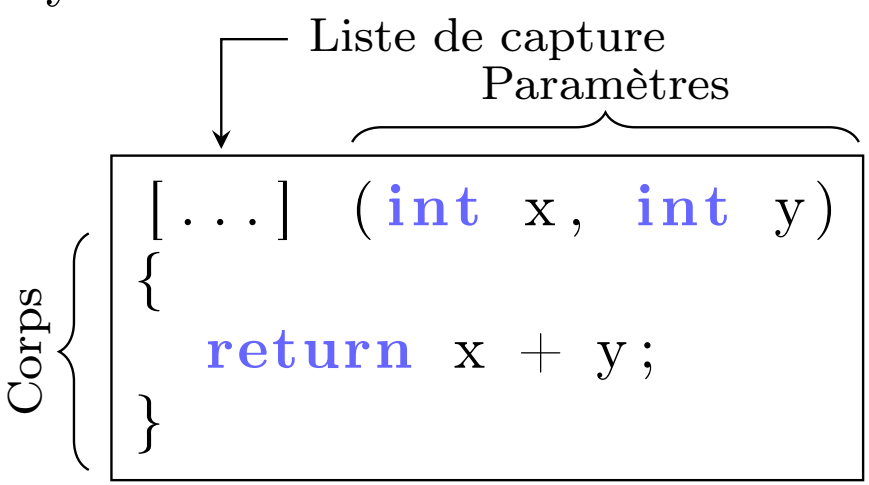
Les lambdas

Idée de base

- Écrire des fonctions (locales) à la volée

- Motivation : la surcharge avec `operator ()` est « trop verbeuse » quand les fonctions sont destinées à un usage unique
- Compilé comme une surcharge d'opérateur

Syntaxe



Exemple

- Fichier `lambda.cpp`

```
1  int main()
2  {
3    vector<int> v = {1, 2, 3, 4, 5};
4    for_each(v.begin(), v.end(), [](int& i) { i++; });
5    for_each(v.begin(), v.end(), [](int i) { cout << i << endl; });
6  }
```

- C'est « court »

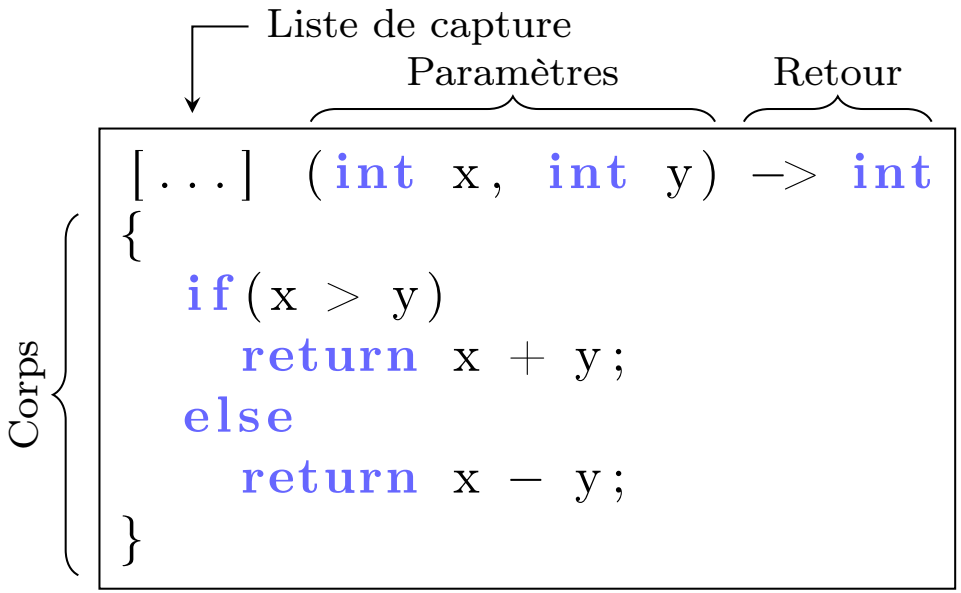
Remarque

- Avoir un code lisible est primordial
- Lambda courtes

Déduction du type de retour

- Dans l'exemple précédent, on n'a pas dû préciser le type de retour
- Il est « déduit » par le compilateur
- Si ce n'est pas possible (si `auto` ne le permet pas), il faut le préciser

Syntaxe



Liste de capture

- Par défaut, rien en dehors des paramètres de la lambda ne peut être utilisé dans son corps
- La liste de capture permet d'inclure des éléments « extérieurs »

Syntaxe

- `[x]` : la variable `x` est passée par valeur
- `[&x]` : la variable `x` est passée par référence
- `[=]` : toutes les variables du bloc de déclaration sont passées par valeur
- `[&]` : toutes les variables du bloc de déclaration sont passées par référence

- Possibilité de combinaison
  - Par exemple : `[x, &y]`

Exemple

- Fichier `lambda-cap.cpp`

```
1  struct A { int i; };
2
3  int main()
4  {
5    A a; a.i = 1;
6    A b; b.i = 2;
7
8    auto f = [&a, b] (int i) //generic lambda
9      {
10       int k = a.i + b.i + i;
11       a.i += 3;
12       //b.i += 3; //error, b is read-only
13
14       return k;
15     };
16
17   cout << f(4) << endl;
18   cout << a.i << " " << b.i << endl;
19 }
```

Initialisation dans la liste de capture

- En C++14, un élément de la liste de capture peut être initialisé

```
1  int main()
2  {
3    int x = 4;
4    auto f = [&r = x, x = x + 1]()->int
5         {//r is a x-reference, x is incremented
6             r += 2;
7             return x + 2;
8         };//();
9
10   int k = f(); //comment that and uncomment stuff before
11
12   cout << x << endl; //6
13   cout << k << endl; //7
14 }
```

Rappel

- Avoir un code lisible est primordial

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

- TU AS FAIT UN NEW !

Question n° 1 : que fait le code suivant ?

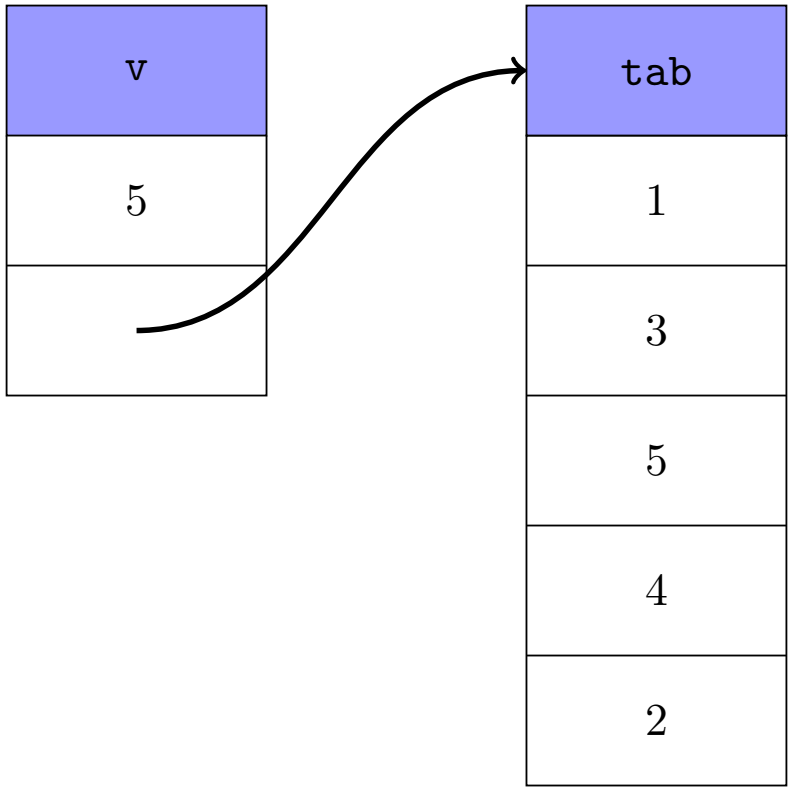
```
1 vector v(5);
2 f(v);
```

Question n° 2 : que fait le code suivant ?

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

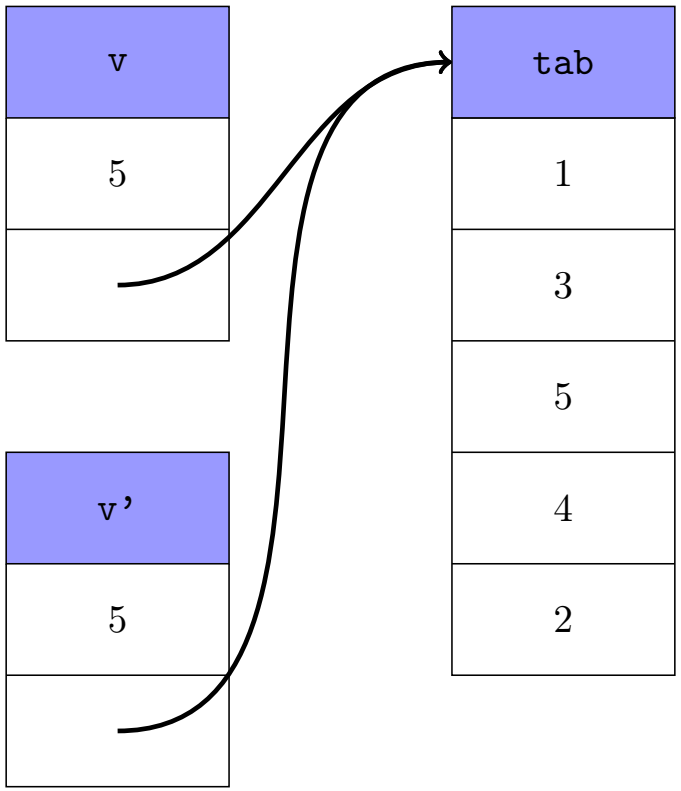
Question n° 1 : appel de `f(v)` (1/3)

- Instanciation du vecteur `v`



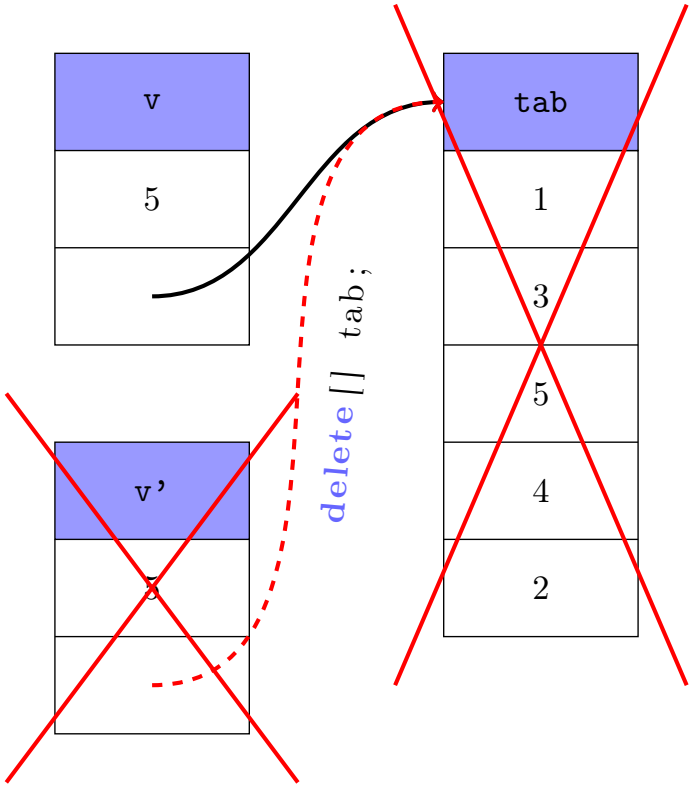
Question n° 1 : appel de `f(v)` (2/3)

- 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

- 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

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

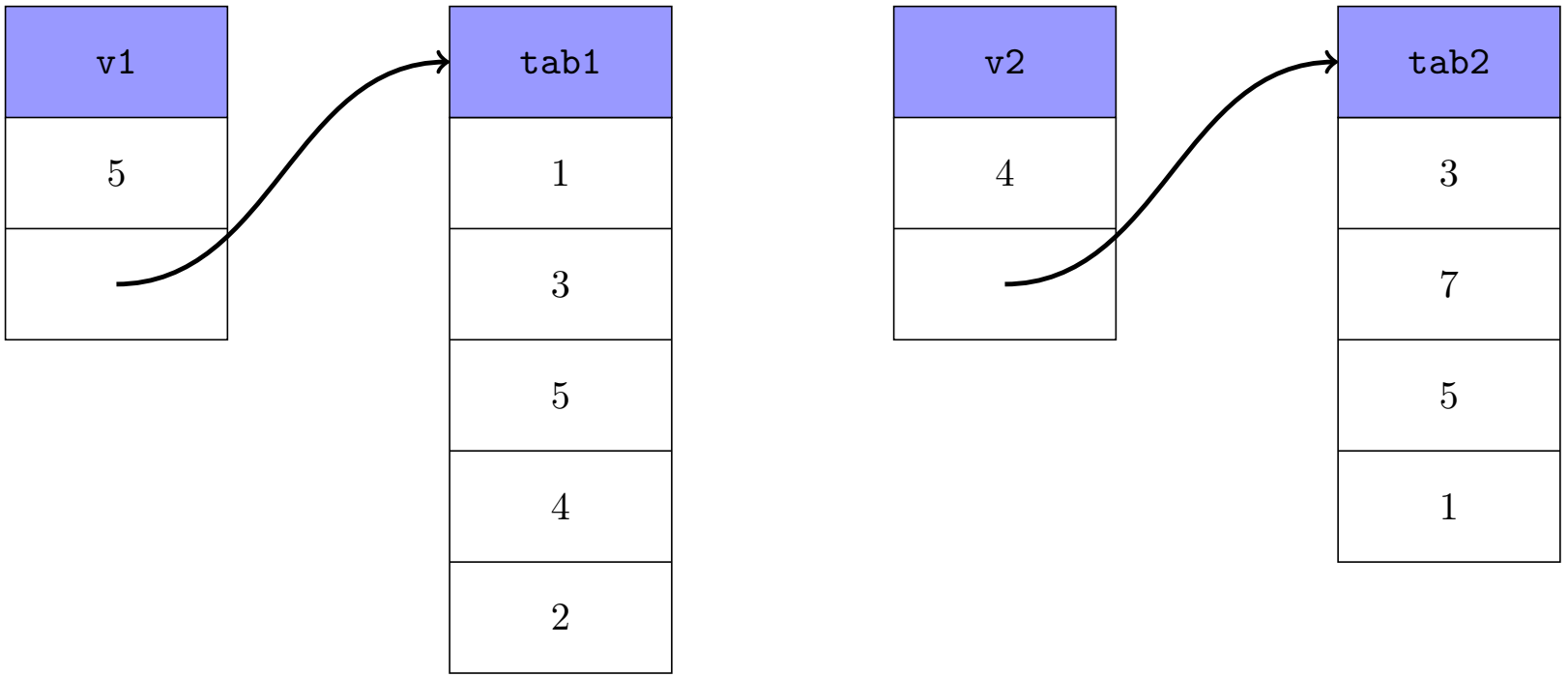
- Fichier `vector-del.cpp`

```
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

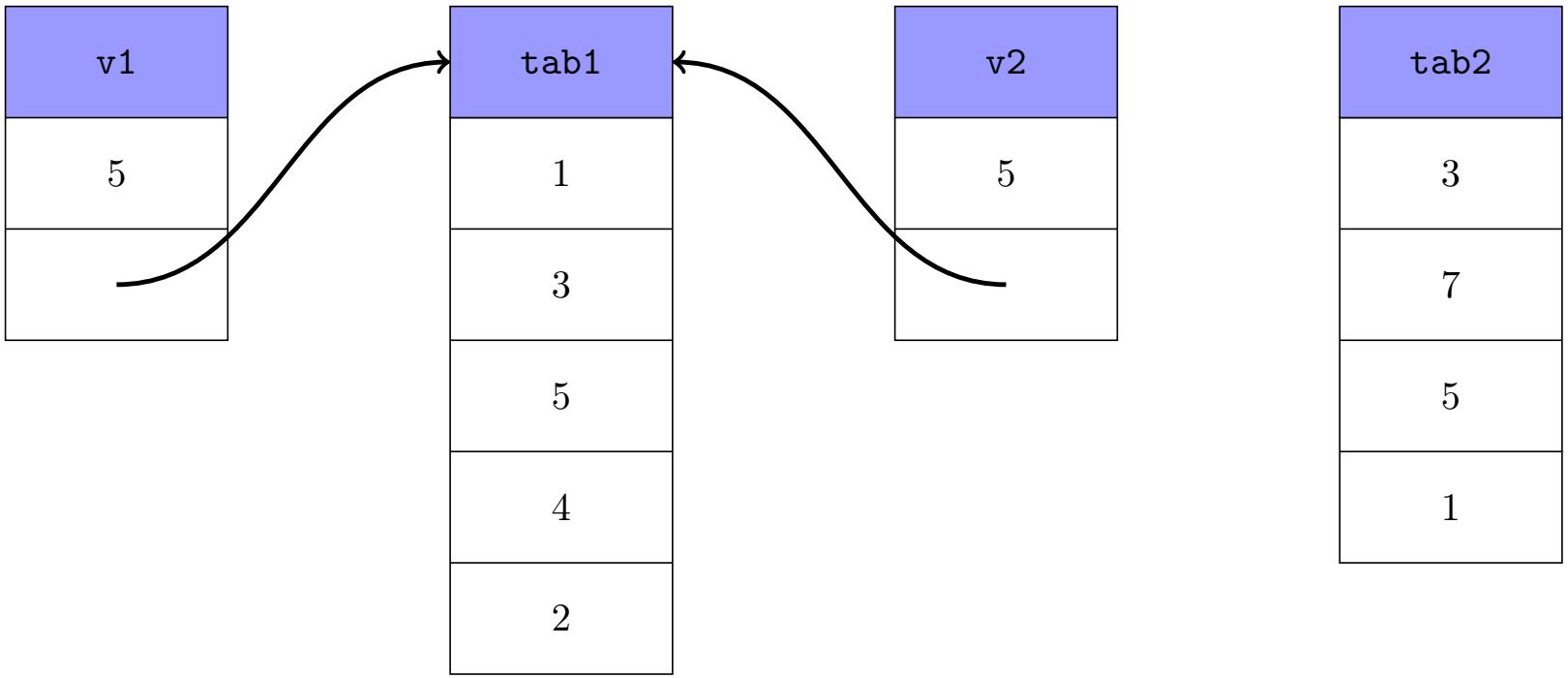
Question n° 2 : affectation de `v1` à `v2` (1/3)

- Instanciation des vecteurs `v1` et `v2`



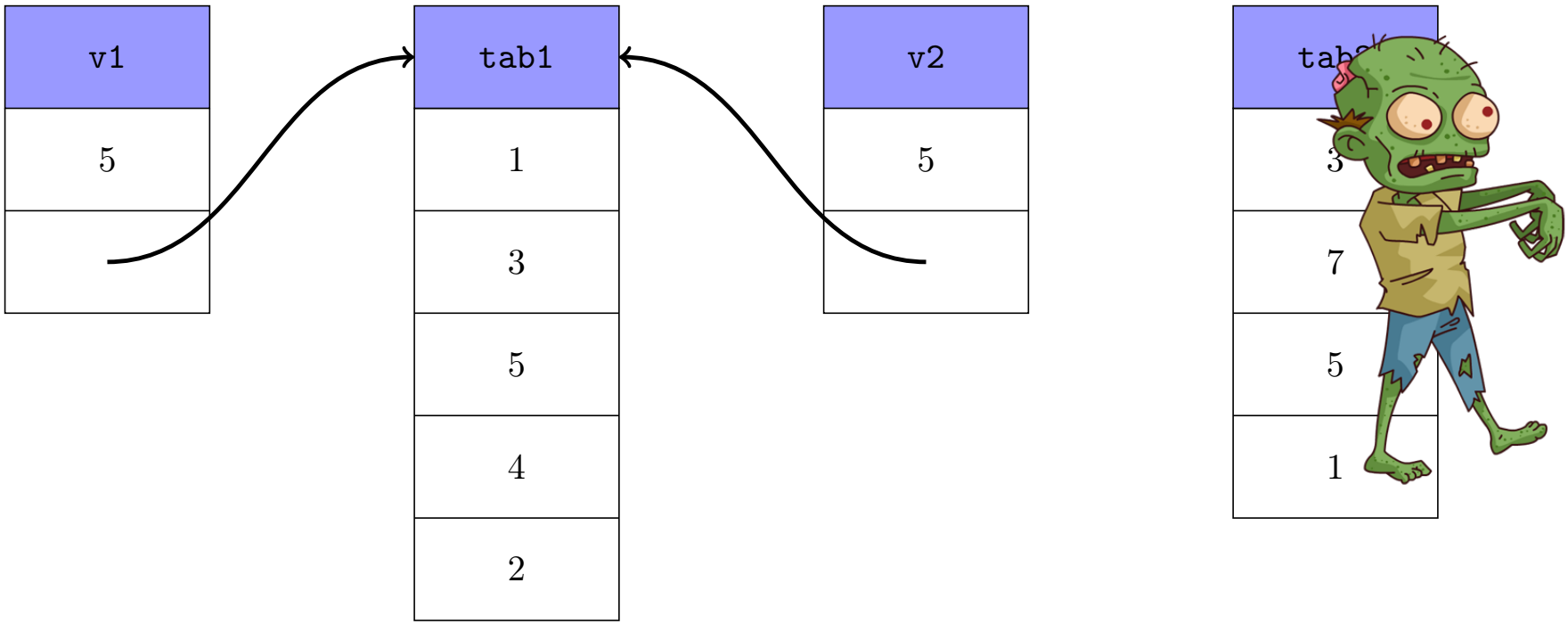
Question n° 2 : affectation de `v1` à `v2` (2/3)

- 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
  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

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 recopie
- 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

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`