M1103 Amphi 3 : Opérateurs, Passage d'arguments

Thomas Nowak

Université Paris-Sud

Opérateurs

Opérateurs

- il est possible de définir les opérateurs qui existent en C++ (comme `+`, `-`, `*`,...) pour nos propres classes
- un opérateur est défini comme une fonction, avec un nom différent
- exemple :

```
Date(2018, 11, 14) + 5
```

- rarement possible dans d'autres langages

Opérateurs

```
Date operator+(Date d, int jours)
{
    for(int i = 0; i < jours; ++i)
        d.rajouter_jour();

    return d;
}
```

Opérateurs

- on ne peut définir que les opérateurs qui existent déjà
- avec leur nombre d'arguments usuel
- doit impliquer au moins un type défini par l'utilisateur (on ne peut pas redéfinir l'addition de `ints`)
- conseil : utiliser seulement si utile

Passage d’arguments

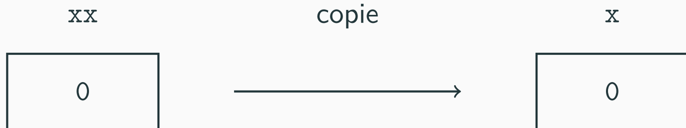
Passage par valeur

- un appel à la fonction

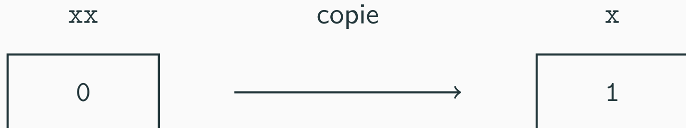
  ```
  int f(int x)
  {
      x = x+1;
      return x;
  }
  ```

  copie la valeur de l'argument dans une variable locale à la fonction `f`
- la valeur de l'argument n'est donc *pas* changée
- l'appel `f(xx)` retourne `xx+1`

Passage par valeur



Passage par valeur



Passage par référence

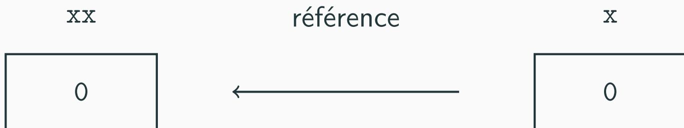
- un appel à la fonction

  ```
  int f(int &x)
  {
      x = x+1;
      return x;
  }
  ```

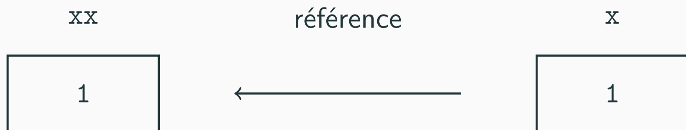
  crée un *alias* de l'argument avec le nom local `x`

- la valeur de l'argument est donc bien changée
- l'appel `f(xx)` retourne `xx+1` et incrémente la valeur de `xx`

Passage par référence



Passage par référence



Passage par référence constante

- on peut déclarer des arguments à une fonction comme *constants*
- possible pour passage par valeur et passage par référence
- pas très utile pour passage par valeur, parce qu'on travaille avec une copie de toute façon
- beaucoup utilisé pour passage par référence
- passage par "référence constante"

Passage d'arguments

- en général, on essaye d'éviter le passage par référence non-constante
- le fait que l'argument puisse changer à l'extérieur peut conduire à des bugs difficiles à trouver
- parfois essentiel
  - changement de plusieurs valeurs
  - changer des collections grandes (ex : vecteurs)

Passage d'arguments

- conseils :
  - passage par valeur pour des objets très petits (∼1-2 `ints`)
  - passage par référence constante pour des objets plus grands
  - passage par référence non-constante seulement si absolument nécessaire
  - c'est souvent mieux de retourner un résultat que de modifier un argument

Range-based for

- on peut parcourir un vecteur avec une syntaxe spécifique, sans utiliser des indices :

```
vector<int> v = {4, 8, 15, 16, 23, 42};

for(int x : v)
    cout << x << endl;
```

Range-based for

- on peut parcourir un vecteur avec une syntaxe spécifique, sans utiliser des indices :

```
vector<string> v = {"hi", "ho", "asdf"};

for(string s : v)
    cout << s << endl;
```

- fait une copie de l'élément courant dans la variable `s`

Range-based for

- il est possible d'écrire :

```
vector<string> v = {"hi", "ho", "asdf"};

for(const string &s : v)
  cout << s << endl;
```

- évite la copie

Namespaces

Namespaces

- pourquoi écrit-on

  ```
  using namespace std;
  ```

  en début de chaque fichier ?
- pour pouvoir écrire `cout` au lieu de `std::cout`
- `std` est un *namespace*, càd une partie du nom “officiel” pour éviter des conflits
- contient `cout`, `cin`, `string`, `vector`, ...

Namespaces

- la déclaration `using` importe tous les élément d'un namespace
- souvent évité dans les fichiers `.h`
- (sinon tout code qui importe le fichier `.h` aura la déclaration `using`)

Retour aux opérateurs

Opérateur <<

- l'opérateur << existe pour plusieurs types :

```
cout << 42 << endl;
cout << 3.14 << endl;
cout << "Hello World!" << endl;
```

- on peut le définir pour nos propres types :

```
ostream& operator<<(ostream &os, const Date &d)
```

- le type de `cout` est `ostream` (ou `std::ostream` officiellement)

Questions