POO & C++ : TP 3

EISE4 2014—2015

Pablo Rauzy

rauzy@enst·fr

pablo.rauzy.name/teaching.html#epu-cpp

13 octobre 2014

N'oubliez pas :

- Les TPs doivent être rendus par courriel à `rauzy@enst·fr` au plus tard le lendemain du jour où ils ont lieu avec `[EISE4]` suivi de vos noms dans le sujet du mail.
- Le code rendu doit être propre !
- Le TP doit être rendu dans une archive tar gzippée : `tar czvf NOMS.tgz NOMS` où `NOMS` est le nom du répertoire dans lequel il y a votre code (idéalement, les noms des gens du groupe) ; attention à ne pas rendre un répertoire avec des fichiers temporaires dus à la compilation (pensez à faire un `make clean` avant `tar`).
- Dans le répertoire avec vos noms, chaque exercice doit être dans son propre répertoire.
- Quand un exercice demande des réponses qui ne sont pas du code, vous les mettrez dans un fichier texte `reponses.txt`.
- Le code est de préférence en anglais, avec des commentaires en français ou anglais, en restant cohérent.
- Le code doit être proprement indenté.
- Respectez les conventions de nommage :
  - variables : `nom_explicite` (*e.g.*, `game_type`),
  - fonctions : `verbeAction` (*e.g.*, `launchNewGame`),
  - classes : `NomExplicite` (*e.g.*, `BoardGame`),
  - membres privés ou protégés : comme variable ou fonction + `_` (*e.g.* `max_number_of_player_`, ou `updateState_`),
  - accesseurs d'affectation : `set_` + nom de l'attribut sans le `_` final (*e.g.*, `set_max_number_of_player`),
  - accesseurs de consultation : nom de l'attribut sans le `_` final (*e.g.*, `max_number_of_player`),
  - constantes : `NOM_EXPLICITE` (*e.g.*, `VERSION_NUMBER`).
- Pensez à utiliser les outils de développement comme le debugger `gdb` ou le profiler `valgrind`.
- N'hésitez jamais à chercher de la documentation par vous-mêmes sur le net !

Dans ce TP :

- Templates
- Exceptions.

Exercice 0.

Récupération des fichiers nécessaires.

1. Pensez à organiser correctement votre espace de travail, par exemple tout ce qui se passe dans ce TP devrait être dans `~/cpp/tp3/`.

2. Récupérez les fichiers nécessaires au TP sur la page du cours.

3. Une fois que vous l'avez extrait de l'archive (`tar xzf tp3.tgz`), renommez le répertoire `tp3` en les noms de votre groupe (par exemple en `Dupont-Dupond`). Si vous ne le faites pas tout de suite, pensez à le faire avant de rendre votre TP.

Exercice 1.

Le but de cet exercice va être de rendre nos conteneurs génériques en plus d'être polymorphiques. Pour cela on va utiliser des templates.

1. Finir rapidement l'exercice 2 du TP 2 (celui qu'on a fait ensemble en classe).

2. Afin de conserver les réponses de l'exercice précédent, copiez les fichiers finaux de l'exercice 2 du TP précédent dans un nouveau répertoire, et travaillez à présent sur ces copies.

→ Transformez la classe abstraite `Iterable` en modèles de classe pour que l'utilisateur puisse choisir un autre type de donnée que des `int`.

Réponse :
Dans `iterable.hpp`, ajouter `template <typename T>` juste avant la déclaration de la classe, et remplacer les `int` par des `T` dans la déclaration.

Dans `iterable.cpp`, les deux fonctions deviennent :

```
template <typename T>
void Iterable<T>::iter (void (*f)(T))
{
  //...
}

template <typename T>
void Iterable<T>::map (T (*f)(T))
{
  //...
}
```

3. → Adaptez le conteneur `Vector` pour le rendre générique.

Réponse :

`vector.hpp`

```
template <typename T>
class Vector : public Iterable<T>
{
public:
  Vector (int size, T *values = NULL);
  Vector (const Vector<T> &v);
  ~Vector ();

  T &operator [] (int i);

  void sort ();
  void begin ();
  bool isAtBeginning ();
  void end ();
  bool isAtEnd ();
  void step ();
  void backstep ();
  T &current ();
  bool isEmpty ();

private:
  T *array_;
  int size_;
  int index_;
};
```

`vector.cpp`

```
template <typename T>
Vector<T>::Vector (int size, T *values) : size_(size) {
  array_ = new T[size_];
  for (int i = 0; i < size_; i++) {
    array_[i] = (values != NULL) ? values[i] : 0;
  }
}
template <typename T>
Vector<T>::~Vector () { delete[] array_; }
template <typename T>
T &Vector<T>::operator [] (int i) {
  if (i < 0 || i >= size_) ERROR
  return array_[i];
}
template <typename T>
void Vector<T>::begin () { index_ = 0; }
template <typename T>
bool Vector<T>::isAtBeginning () { return index_ == 0; }
template <typename T>
void Vector<T>::end () { index_ = size_ - 1; }
template <typename T>
bool Vector<T>::isAtEnd () { return index_ == size_ - 1; }
template <typename T>
void Vector<T>::step () { if (index_ < size_ - 1) index_++; }
template <typename T>
void Vector<T>::backstep () { if (index_ > 0) index_--; }
template <typename T>
T &Vector<T>::current () { return array_[index_]; }
template <typename T>
bool Vector<T>::isEmpty () { return false; }
```

4. Pour le conteneur `List`, il faut aussi penser à mettre à jour la classe `Node` pour en faire un modèle aussi.

→ Adaptez le conteneur `List` pour le rendre générique.

Réponse :
Même idée. Ne pas oublier de rendre `Node` générique aussi et dedans de déclarer `friend class List<T>;` plutôt que `friend class List;`.

5. Pour rappel, la dernière question du TP 2 était : "*Écrivez vous-même une nouvelle fonction dans le fichier `main.cpp` qui utilise l'interface `Iterable` pour itérer sur les éléments en ordre inverse et n'afficher que les multiples d'un nombre qu'on lui a passé en argument, puis testez-la pour les `List` et pour les `Vector`.*".

On suppose qu'on a rendu cette fonction générique aussi en en faisant un modèle de fonction qui prend non plus un `Iterable` mais un `Iterable<T>` en argument.

→ Sans nécessairement coder, quels problèmes pensez-vous rencontrer en essayant par exemple de déclarer un `List<Vector<string> >` et d'utiliser la fonction de la dernière question du TP 2 dessus ?

Sans nécessairement le faire non plus, dites comment vous régleriez ces problèmes.

Réponse :
La fonction du TP 2 fait appel aux opérateurs `%` et `<<` sur les objets contenu dans l'`Iterable` qu'on lui passe en paramètre. Cela ne posait pas de problème avant car ces objets était forcément des entiers. Mais dans notre cas ces objets seront de type `Vector<string>`.
Pour résoudre ces problèmes il faut surcharger les opérateurs `<<` et `%` dans le modèle de classe `Vector<T>`. Le premier pour l'affichage dans un `ostream` et le second pour par exemple renvoyer le modulo avec la taille du `Vector`.

Exercice 2.

Dans cet exercice le code d'une calculatrice en notation polonaise inversée vous est fourni, mais il ne gère pas les erreurs. Le but de l'exercice va être de rajouter la gestions des erreurs en utilisant des exceptions.

Le principe de la notation polonaise inversée est de fonctionner avec une pile de valeurs et de faire les opérations en dépilant deux valeurs et en rempliant le résultat.

Par exemple, le calcul `1 + ((2 + 3) * 4 - 5)` s'écrit `1 2 3 + 4 * 5 - +`. Cela se lit comme :

- empile `1` ; `[ 1 ]`
- empile `2` ; `[ 2 1 ]`
- empile `3` ; `[ 3 2 1 ]`
- dépile `3`, dépile `2`, empile `2 + 3 = 5` ; `[ 5 1 ]`
- empile `4` ; `[ 4 5 1 ]`
- dépile `4`, dépile `5`, empile `5 * 4 = 20` ; `[ 20 1 ]`
- empile `5` ; `[ 5 20 1 ]`
- dépile `5`, dépile `20`, empile `20 - 5 = 15` ; `[ 15 1 ]`
- dépile `15`, dépile `1`, empile `1 + 15 = 16`. `[ 16 ]`

1. Explorez le code dans le but de comprendre comment il est organisé et comment il fonctionne.

→ Décrivez brièvement ce que vous en avez compris : quel est le rôle les différents modules, quelles exceptions existent déjà et à quoi servent-elles. Donnez au passage la liste des opérations qui sont possibles.

Réponse :
Il y a 4 modules : `REPL`, `Stack`, `Expression`, et `Exceptions`.
Le module `Exceptions` contient les déclarations des exceptions.
Le module `Expression` représente les expressions qui peuvent être entrée par l'utilisateur : des nombres et des opérateurs. Les opérateurs implémentés sont l'addition, la soustraction, la multiplications, la division, et le modulo, sur les entiers.
Le module `Stack` gère la pile d'entier.
Le module `REPL`, pour "read-eval-print-loop" est le module principal du programme, il contient et gère la pile de la calculatrice. C'est lui qui se charge de lire les entrées utilisateurs (read), de les évaluer, et d'afficher la pile en retour.

2. Compilez le programme en utilisant le `Makefile` fourni, puis jouez un peu avec la calculatrice pour comprendre comment elle fonctionne.

→ Normalement, un avertissement concernant un `switch` est affiché par le compilateur lors de la compilation, pouvez-vous l'expliquer (et faites le) ? Le corriger (comment ?) ?

Réponse :
Dans la fonction `REPL::eval`, il y a un `switch` sur le type de l'expression, et il manque un cas possible (`Expression::Number`).
Il y a plusieurs correction possible. Le plus propre est sans doute de rajouter une clause `default` qui lance un `exit(EXIT_FAILURE);` ou qui lève une exception `ShouldNeverHappen` par exemple.

3. Il y a un autre `switch` pour lequel le compilateur ne peut pas afficher le même genre d'avertissement et où il serait pourtant bien utile.

→ Pouvez-vous dire de quel `switch` il s'agit ? Décrivez comment déclencher une erreur lors de l'exécution du programme en utilisant le code problématique.

Réponse :
Il manque une clause `default` dans le `switch` de la fonction `REPL::read` et donc quand l'utilisateur entre un opérateur non prévu, on retourne un pointeur `NULL`, ce qui provoque une erreur de segmentation lors du déréférencement pour l'appel à l'accesseur `Expression::type` dans `REPL::eval`.

4. → Ajoutez une exception pour gérer ce problème et utilisez là faire un retour à l'utilisateur.

Réponse :
Dans `exceptions.hpp` :

```
#include <sstream>
class UnknownOperator : public std::exception
{
public:
  UnknownOperator (char op)
    : op_(op)
  {}

  const char *what () const throw()
  {
    std::stringstream err;
    err << "Operator '" << op_ << "' unknown.";
    return err.str().c_str();
  }

private:
  char op_;
};
```

Dans le `switch` de la fonction `REPL::read` dans `REPL.cpp` :

```
default:
  throw new UnknownOperator(op);
```

Après le dernier bloc `catch` dans la fonction `REPL::loop` :

```
catch (UnknownOperator *exn) {
  err_ << "ERROR: " << exn->what() << std::endl;
  delete exn;
  continue;
}
```

Ou on peut aussi modifier le dernier bloc `catch` pour qu'il attrape les exceptions de type `std::exceptions *` plutôt que seulement les `EmptyStack *`, en utilisant le polymorphisme.

5. Il reste encore d'autres soucis. Par exemple, regardez ce qu'il se passe quand on entre le calcul `1 2 3 + - *`. Ce n'est pas satisfaisant : la pile n'était pas vraiment vide, elle ne contenait qu'un seul élément, que l'on perd complètement du coup.

→ Corrigez ce problème à l'aide d'une nouvelle exception `StackTooSmall`.

Réponse :
Dans `exceptions.hpp` :

```
class StackTooSmall : public std::exception
{
public:
  const char *what () const throw()
  {
    return "Not enough element on stack.";
  }
};
```

Au début de la branche `else` de la fonction `REPL::eval` :

```
if (stack_->size() < 2) {
  throw new StackTooSmall;
}
```

Après le dernier bloc `catch` dans la fonction `REPL::loop` :

```
catch (StackTooSmall *exn) {
  err_ << "ERROR: " << exn->what() << std::endl;
  delete exn;
  continue;
}
```

Il n'y a pas besoin de cette dernière modification si la solution polymorphique a été utilisée à la question précédente.

6. Il reste une erreur possible, qui est très classique, avec la division.

→ Corrigez la avec une nouvelle exception, qui cette fois fera quitter le programme après avoir afficher l'erreur à l'utilisateur.

Réponse :
Il s'agit évidemment de la division par zéro.
Exemple de gestion :
Dans `exceptions.hpp` :

```
class DivisionByZero {};
```

Dans le `case Expression::DivOperator:` du `switch` de la fonction `REPL::eval` :

```
if (a == 0) {
  throw new DivisionByZero;
}
```

Dans `REPL::lopp` :

```
catch (CtrlD *exn) {
  out_ << "FATAL ERROR: Division by zero." << std::endl;
  delete exn;
  break;
}
```