

La programmation orientée objet et le langage C++

Pablo Rauzy

rauzy@enst.fr

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

Ce cours est en partie inspiré du polycopié d'Henri Garreta.

EISE4 @ Polytech'UPMC

6 octobre 2014 — Cours 3

Cours précédents

Modèles

Exceptions

TP 2

- ▶ Nouveautés du C++ par rapport au C.
 - ▶ Déclarations, `bool`, références, `inline`, arguments par défaut, surcharge des fonctions, I/O, `new/delete`.
- ▶ Programmation orientée objet, encapsulation.
- ▶ Les classes en C++.
 - ▶ Syntaxe, politique d'accès aux membres, programmation modulaire, constructeurs, destructeurs, membres constants, membres statiques, amie.
- ▶ Surcharge des opérateurs.
 - ▶ Surcharge des fonctions membres et non-membres, conversions de types.
- ▶ Héritage.
 - ▶ Accessibilité et politiques de dérivation, redéfinition, constructions et destructions des instances de classes dérivées, polymorphisme, fonction virtuelles (pures), classes abstraites, conversions et identifications dynamiques.

- ▶ Un modèle, ou *templates*, définit une famille de fonctions ou de classes paramétrée par des types et des valeurs.
- ▶ Un modèle est introduit par le mot clef `template` suivi de la liste de ses paramètres entre chevrons (`<` et `>`).
- ▶ Les paramètres peuvent soit être des types (introduits par le mot clef `typename`) soit des valeurs (introduites par leur type, comme des arguments classiques).
- ▶ Les modèles sont *instanciés* par le compilateur, c'est à dire qu'il produit le code spécialisé pour chaque ensemble de paramètres utilisés.
- ▶ Même principe que les *macros* mais bien plus complexe.

- ▶ Syntaxe :

`template<paramètres> déclaration fonction.`

- ▶ Par exemple :

- ▶ déclaration :

```
template<typename T>
    T minimum (T array[], int size) { ... },
```

- ▶ utilisations :

```
coldest = minimum<double> (temp_per_month, 12);
a = minimum<char> (randomized_alphabet, 26);
```

- ▶ Un modèle plus spécialisé (avec moins de paramètres) pour la même fonction se substitue toujours aux autres quand il est applicable (simple surcharge).
- ▶ À l'utilisation on peut omettre de préciser les paramètres si ils peuvent se déduire du type des arguments.

```
template<typename T>
T minimum (T array[], int size)
{
    T min = array[0];
    for (int i = 1; i < size; i++) {
        if (array[i] < min)
            min = array[i];
    }
    return min;
}

double avg_temp_per_month[12] = { 7.2, 8.1, 11.0, 13.9, 18.0, 21.9,
                                24.8, 24.4, 20.6, 16.7, 11.2, 8.0 };

double min_avg = minimum<double>(avg_temp_per_month, 12);

Volt voltages[3] = { Volt(220), Volt(5.5), Volt (12) };

Volt min = minimum<Volt>(voltages, 3);
// on suppose ici que Volt surcharge l'opérateur < pour la comparaison
```

```
char *minimum (char *array[], int size) {
    char *min = array[0];
    for (int i = 1; i < size; i++) {
        if (strcmp(array[i], min) < 0)
            min = array[i];
    }
    return min;
}

double avg_temp_per_month[12] = { 7.2, 8.1, 11.0, 13.9, 18.0, 21.9,
                                24.8, 24.4, 20.6, 16.7, 11.2, 8.0 };

double min_avg = minimum(avg_temp_per_month, 12);

char *fruits[] = { "banana", "kiwi", "avocado", "tomato", "peach",
                  "pear", "orange", "apricot", "pineapple" };

char *first_alpha = minimum(fruits, 9);
```

- ▶ Un modèle de classe est un type paramétré par d'autres types et par des valeurs connues lors de la compilation.
- ▶ Les paramètres des modèles de classes doivent toujours être explicités.

- ▶ Les fonctions membres d'un modèles de classes sont des modèles de fonctions avec les mêmes paramètres.
- ▶ C'est implicite quand les fonctions sont définies dans la déclaration de la classe.
- ▶ Il faut l'explicitier quand les fonctions sont définies en dehors.

```
template <typename Elements_t>
Elements_t &Vector<Elements_t>::operator [] (int i)
{
    if (i < 0 || i >= size_) {
        cerr << "Error: [Vector<" << typeid(Elements_t).name()
              << ">] index out of bound (" << i << ')' << endl;
        exit(EXIT_FAILURE);
    }
    return array_[i];
}
```

```
template <typename Elements_t>
class Vector
{
public:
    Vector (int size)
        : size_(size)
    {
        array_ = new Elements_t[size_];
    }

    ~Vector ()
    {
        delete[] array_;
    }

    Elements_t &operator [] (int i)
    {
        if (i < 0 || i >= size_) {
            cerr << "Error: [Vector<" << typeid(Elements_t).name()
                  << ">] index out of bound (" << i << ')' << endl;
            exit(EXIT_FAILURE);
        }
        return array_[i];
    }

private:
    Elements_t *array_;
    int size_;
};

Vector<string> names(28);
Vector<Shape> drawing(10);
```

- ▶ Comme pour les arguments des fonctions, les modèles peuvent avoir des paramètres par défaut.

```
template <typename Elements_t = int, int vector_size = 10>
class FixedVector
{
public:
    Elements_t &operator [] (int i)
    {
        if (i < 0 || i >= vector_size) {
            cerr << "Error: [Vector<" << typeid(Elements_t).name()
                  << ">] index out of bound (" << i << ')' << endl;
            exit(EXIT_FAILURE);
        }
        return array_[i];
    }

private:
    Elements_t array_[vector_size];
};

FixedVector<double, 32> a; // tableau de 32 double
FixedVector<double> b; // tableau de 10 double
FixedVector<> c; // tableau de 10 int
FixedVector d; // ERREUR
```

- ▶ Tout comme on peut déclarer l'existence d'une classe, on peut déclarer l'existence d'un modèle.

```
template<typename Elements_t> class Vector;

Vector<string> *days; // ok, l'existence suffit
Vector<string> months; // ERREUR
```

- ▶ Un paramètre de modèle peut être un modèle puis même, mais attention à la syntaxe.

```
FixedVector< Vector<int> * > foo; // tableau de 10 pointeurs sur des tableaux d'entiers
```

- ▶ La fonction qui détecte l'anomalie *lance* (`throw`) une exception.
- ▶ L'exception est une valeur d'un type quelconque (souvent une classe dérivée de `exception`).
- ▶ Elle *traverse* la pile d'appels jusqu'à atteindre une fonction qui a prévu d'*attraper* (`catch`) ce type d'exception.
- ▶ Les fonctions qui sont traversées (y compris celle qui a lancée l'exception) sont terminées sur le champ (les instructions non encore exécutées sont abandonnées), mais les objets locaux sont détruits.
- ▶ Si une exception traverse toute la pile d'appels sans jamais être attrapée, le programme termine.

- ▶ Parfois, une fonction de bas niveau (dans une bibliothèque) peut détecter une anomalie, mais ne peut pas décider à la place du programmeur comment y réagir.
- ▶ C'est à ça que serve les *exceptions*.

- ▶ On lance une exception avec `throw expression`.
- ▶ On l'attrape en utilisant un *bloc try ... catch*.

```
try {
    // instructions susceptibles de lancer une exceptions
    // (directement ou dans les fonctions appelées)
}
catch (type1 e1) {
    // code pour traiter une exception de type type1
}
catch (type2 e2) {
    // code pour traiter une exception de type type2
}
catch (...) {
    // code pour les autres exceptions
}
```

- ▶ Un bloc `try` doit toujours être suivi d'un ou plusieurs blocs `catch`.
- ▶ Le dernier bloc `catch` peut servir de "pokemon exception handling" (*gotta catch 'em all*) en utilisant la syntaxe `...`
- ▶ Dès qu'on entre dans un bloc `catch`, l'exception est considérée comme traitée; si il y a besoin de la propager plus loin cela peut être fait avec l'instruction `throw`;

- ▶ Quand une exception est lancée, un bloc `catch` compatible avec l'exception est recherché en remontant la pile d'appels.
- ▶ L'argument d'un bloc `catch` est de type `T1`, `const T1`, `T1 &`, ou `const T1 &` est compatible avec une exception de type `T2` si et seulement si :
 - ▶ `T1` et `T2` sont le même type,
 - ▶ `T1` est une classe de base de `T2`, ou
 - ▶ `T1` est un pointeur sur une classe de base accessible de celle vers laquelle `T2` pointe.

```
template <typename Elements_t>
Elements_t &Vector<Elements_t>::operator [] (int i)
{
    if (i < 0 || i >= size_) {
        throw "Index out of bounds";
    }
    return array_[i];
}

//...

try {
    Vector<int> a[5];
    cout << a[6] << endl;
}
catch (char *exn) {
    cerr << "Caught exception '" << exn << "'." << endl;
}
```

```
class BadIndex : public exception {...};
class DivisionByZero : public exception {...};
class InvalidArgument : public exception {...};

try {
    //...
}
catch (exception *e) {
    cerr << e->what() << endl;
}
```



- ▶ Par défaut, le compilateur suppose que les fonctions se laissent traverser par tout type d'exceptions.
- ▶ Une fonction peut indiquer le type d'exception qu'elle laisse échapper.
- ▶ Cela peut servir de documentation dans le code, et aussi à générer des avertissements du compilateurs si cette contrainte n'est pas respectée.
- ▶ Syntaxe :
function prototype `throw (type1, type2, ...)`.
- ▶ avec simplement `throw ()` on indique qu'une fonction ne doit lancer aucune exception.
- ▶ La déclaration `throw (...)` ne fait pas partie de la signature de la fonction.

- ▶ Reprise des notions du cours précédent sur l'héritage en faisant le second exercice du TP 2 en classe.

```
class exception {
public:
    exception () throw ();
    exception (const exception &e) throw ();
    virtual ~exception () throw ();

    exception &operator = (const exception &e) throw ();
    virtual const char *what () const throw ();
};
```

Cours précédents

Modèles

- Modèles de fonctions
- Spécialisation
- Modèles de classes
- Fonctions membres d'un modèle de classes
- Paramètres par défaut des modèles
- Dernières petites choses

Exceptions

- Mécanisme
- Syntaxe
- Principes
- Exemple : exceptions
- Attraper une exception
- Déclaration des exceptions qu'une fonction laisse échapper
- La classe `exception`

TP 2