Programmation orientée objet

Fonctions et classes génériques

Samuel Kadoury

Fonctions génériques

- Il arrive souvent d'être confronté à deux codes identiques à une exception près:

  le **type** des entités manipulées

- *Polymorphisme ad hoc* (surcharge) requiert d'écrire plusieurs fois le même code
- *Polymorphisme par sous-typage* ne marche que pour des types dérivant du même ancêtre
- Solution: ***fonctions génériques*** *(ou modèles de fonction, ou patrons de fonction)*

Exemple de fonction générique

- Soit une fonction qui affiche toutes les valeurs d'un tableau:

Il faudrait spécifier un type ici.

```
void print(ostream& out, ??? data[], int count)
{
  out << "[";
  for (int i = 0; i < count; ++i) {
    if (i > 0)
       out << ",";
    out << data[i];
  }
}
```

Exemple de fonction générique (suite)

Quelques fois on voudrait passer un tableau de chaînes de caractères.

```
void print(ostream& out, string data[], int count)
{
  out << "[";
  for (int i = 0; i < count; ++i) {
    if (i > 0)
      out << ",";
    out << data[i];
  }
}
```

Exemple de fonction générique (suite)

D'autres fois on voudrait passer un tableau d'entiers.

```
void print(ostream& out, int data[], int count)
{
  out << "[";
  for (int i = 0; i < count; ++i) {
    if (i > 0)
      out << ",";
    out << data[i];
  }
}
```

Exemple de fonction générique (suite)

... ou même un type d'objet défini par nous.

```
void print(ostream& out, Employee data[], int count)
{
  out << "[";
  for (int i = 0; i < count; ++i) {
    if (i > 0)
      out << ",";
    out << data[i];
  }
}
```

Définition de fonction générique

- On précède la fonction d'une déclaration ayant la forme suivante:

```
template< typename T1, typename T2, ..., typename T3 >
```

où T1, T2, .. T3 sont des paramètres représentant des types

Exemple de fonction générique

- Notre exemple sera donc écrit de la manière suivante:

```
template< typename T >
void print(ostream& out, T data[], int count)
{
  out << "[";
  for (int i = 0; i < count; ++i) {
    if (i > 0)
      out << ",";
    out << data[i];
  }
}
```

Exemple de fonction générique (suite)

- Exemple d'utilisation de notre fonction générique:

```
int main()
{
 string a[] = {"Jorge", "Georges", "George"};
 int b[] = {2, 4, 5, 8};
 print(cout,a,3);
 print(cout,b,4);
    ...
```

Le compilateur créera deux versions de la fonction: une pour les *string* et une autre pour les *int*.

Polymorphisme de compilation

- Une fonction générique est donc une fonction qui accepte plusieurs types de paramètre
- Quel que soit le type de paramètre qui lui est passé, elle fait exactement le même traitement
- En plus du terme *polymorphisme paramétré*, cette technique est appelée *polymorphisme de compilation*
- Cela signifie qu'à la compilation, le **compilateur crée autant de versions de la fonction que nécessaire**

Le compilateur trouve le type correct “automatiquement”

```
template<typename T>
void f(){
   ...
}

int main(){
   f();// ERREUR: valeur de T?
   f<int>();// choix explicit de T
}
```

Contrainte câchée sur les types

- Attention, il y a une condition pour qu'on puisse appeler une fonction générique
- **Dans la fonction, toutes les opérations effectuées sur un paramètre de type générique doivent être valides pour le type en question**

Contrainte câchée sur les types (exemple)

```
template< typename T >
T maximum(const T& left, const T& right)
{
   if (left < right)
    return right;
   return left;
}
```

Si ces deux paramètres sont des objets, il faut que l'opérateur < soit défini pour leur classe

Classes génériques

- Il arrive que des classes soient identiques, à l'exception des types des attributs, ou encore des types des paramètres des méthodes
- Exemple de classe générique: les listes!

Liste liée générique

```
template< typename T >
class Node
{
public:
    Node(T s);
private:
    T data;
    Node<T>* previous_;
    Node<T>* next_;
    ...
};
```

Liste liée générique (suite)

```
template< typename T >
class Iterator
{
public:
    Iterator();
    T get() const;
    void next();
    bool equals(Iterator<T> iter) const;
private:
   Node<T>* position;
};
```

Liste liée générique (suite)

```
template< typename T >
class List
{
public:
    List();
    void push_back(T s);
    void insert(Iterator<T> pos, T s);
    Iterator<T> erase(Iterator<T> pos);
    Iterator<T> begin();
    Iterator<T> end();
private:
    Node<T>* first_;
    Node<T>* last_;
};
```

Liste de STL

●Pas besoin de définir une classe List, au meme titre que Vector, puisqu'il y en a déjà une fournie dans la bibliothèque STL:

- Il faut faire **`#include <list>`** pour inclure le fichier d'en-tête
- On utilise alors la class **`list`**, qui est une classe générique à un paramètre de type

Liste de STL (suite)

- Principales méthodes définies pour cette classe:
  - push_back(), pour ajouter un item à la fin
  - push_front(), pour ajouter un item au début
  - pop_back(), pour retirer le dernier item de la liste
  - pop_front(), pour retirer le premier iterm de la liste
  - front(), qui retourne le premier élément de la liste
  - back(), qui retourne le dernier élément de la liste
  - size(), pour connaître le nombre d'items dans la liste
  - empty(), pour vérifier si la liste est vide
- Les itérateurs de liste seront vus plus tard lorsque nous présenterons la bibliothèque STL

Liste de STL (exemple)

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    int valeur;
    list<int> liste;
    cin >> valeur;

    // On lit une liste d'entiers
    while (!cin.eof()) {
        liste.push_front(valeur);
        cin >> valeur;
    }
    // On les affiche en ordre inverse
    while (!liste.empty()) {
        cout << liste.front() << endl;
        liste.pop_front();
    }
     return 0;
}
```

Une autre classe générique: Pair

- Représente une paire de valeurs, pas nécessairement du même type
- Pas de méthodes, mais deux attributs public:
  - first
  - second
- Exemple:

```
Pair< int, string > une_paire(4,"quatre");
cout << pair.first << " "  << pair.second;
```

Implémentation de Pair

```
template< typename F, typename S >
class Pair
{
 public:
    Pair(const F& a, const S& b);
    F getFirst() const;
    S getSecond() const;
 private:
    F first_;
    S second_;

};
```

Voici un exemple de classe qui n'a pas de constructeur par défaut.

Ces deux attributs peuvent être de n'importe quel type. Ils peuvent aussi être du même type ou de types différents.

Exemple de la classe Pair (suite)

Il faudra répéter cette déclaration pour chaque fonction membre définie dans la classe.

```
template< typename F, typename S >
Pair<F, S>::Pair(const F& a, const S& b)
: first_(a), second_(b)
{
}
```

Utilisation d'une classe générique

Contrairement aux fonctions génériques!

- **Chaque fois qu'on déclare un objet d'une classe générique, il faut toujours spécifier tous les types qui sont paramétrisés dans le modèle**
- Par exemple, on déclare de la manière suivante une paire de deux entiers:

```
Pair< int, int > unePaire(3,4);
```

- Pour déclarer une paire composée d'un entier et d'une chaîne de caractères :

```
Pair< int, string > unePaire(3,"Michel");
```

L'mplémentation d'une classe générique doit être dans le .h!

```
#ifndef PAIR_H
#define PAIR_H
   template< typename F, typename S >
   class Pair{
    public:
       Pair(const F& a, const S& b);
       …
    private:
       …
   };

   template< typename F, typename S >
   Pair<F, S>::Pair(const F& a, const S& b)
   : first_(a), second_(b){}
   …
#endif
```

Pourquoi? Quand le compilateur voit l'utilisation d'une classe générique, il doit avoir accès à la définition et l'implémentation pour produire une version spécifique de la classe!

Argument non type

- Dans certains cas, on veut spécifier non pas un type variable, mais une valeur variable
- Exemple d'une classe matrice:

```
template< typename T, int ROWS, int COLUMNS >
class Matrix
{
...
private:
 T data_[ROWS][COLUMNS];
};
```

Argument non type (suite)

- Ainsi, on pourra déclarer des matrices de types différents et de tailles différentes:

```
Matrix< int, 3, 4 > a;
Matrix< double, 3, 4 > b;
Matrix< string, 2, 3 > c;
```