Programmation Orientée Objet

Structures de données abstraites et généricité en C++

Guillaume Revy

guillaume.revy@univ-perp.fr

Université de Perpignan Via Domitia

Plan du cours

1. Structure de données abstraites

2. Vers la généricité de ces structures
   - Rappel de ce qu'est un template
   - Modèle de fonctions/méthodes
   - Modèle de classes

Plan du cours

1. Structure de données abstraites

2. Vers la généricité de ces structures
   - Rappel de ce qu'est un template
   - Modèle de fonctions/méthodes
   - Modèle de classes

Qu'est-ce qu'une structures de données abstraites ?

- Une structure de données abstraite est un ensemble organisé d'information (c'est-à-dire, de données) reliées logiquement et pouvant être manipulées individuellement mais également comme un tout.
  - modéliser au mieux les informations à traîter et faciliter leurs traitements
- Son utilisation (via les fonctions de manipulation associées) facilite énormément l'implémentation logicielle d'un algorithme
  - choix des structures de données $\rightsquigarrow$ aussi important que celui de l'algorithme

Qu'est-ce qu'une structures de données abstraites ?

- Une structure de données abstraite est un ensemble organisé d'information (c'est-à-dire, de données) reliées logiquement et pouvant être manipulées individuellement mais également comme un tout.
  - modéliser au mieux les informations à traîter et faciliter leurs traitements
- Son utilisation (via les fonctions de manipulation associées) facilite énormément l'implémentation logicielle d'un algorithme
  - choix des structures de données $\rightsquigarrow$ aussi important que celui de l'algorithme
- Exemple : un tableau $\rightsquigarrow$ ensemble d'éléments de type donné
  - connaître la taille du tableau, accéder en lecture/écriture aux éléments du tableau, redimensionner le tableau, trier les éléments du tableau, ...
  - autres exemples : liste simplement/doublement chaînée, pile, file, vecteur, tableau associatif, ... $\rightsquigarrow$ disponibles en C++ dans la STL ! !

Spécialisations des structures de données abstraites ?

- Une structure de données abstraite est caractérisée par :
  - son contenu,
  - et les interactions possibles (manipulations, accès, ...)

- Du point de vue informatique, une structure de données abstraite peut être spécifiée à plusieurs niveaux :
  - fonctionnel/logique $\rightsquigarrow$ spécification formelle des données et algorithmes de manipulation associés
    - généralisation de la notion d'*objets manipulables*, sans connaître sa structure interne, ni son implantation
  - physique/programmation $\rightsquigarrow$ implantation de la structure données en mémoire, dans la machine
    - déterminant pour l'efficacité des programmes utilisant ces structures

Exemples de structures de données abstraites

- Les conteneurs de la STL (cf. Cours 06) :
  - les paires (`pair`), les vecteurs (`vector`), les listes doublement chaînées (`list`), les tableaux dynamiques (`deque`), les ensembles ordonnés (`set`), les tables associatives ordonnées (`map`), les piles (`stack`), les files (`queue`), ...

  mais également d'autres structures :

  - les arbres, les graphes, les multi-listes, ...

Exemples de structures de données abstraites

- Les conteneurs de la STL (cf. Cours 06) :
  - les paires (`pair`), les vecteurs (`vector`), les listes doublement chaînées (`list`), les tableaux dynamiques (`deque`), les ensembles ordonnés (`set`), les tables associatives ordonnées (`map`), les piles (`stack`), les files (`queue`), ...

  mais également d'autres structures :
  - les arbres, les graphes, les multi-listes, ...

- Et pour chaque structure de données (ie. les tableaux), on a plusieurs services :
  - les constructueurs *construire*
  - les itérateurs $\rightsquigarrow$ `[i]` *parcourir*
  - les sélecteurs $\rightsquigarrow$ `t[i]`, `t.size()` *interroger*
  - les modificateurs $\rightsquigarrow$ `t[i] = i` *modifier*

Définition de la classe Tableau

- Définition d'un tableau d'entiers (cf. TD2)

```
class Tableau
{
private:
  int * elts;
  unsigned int current_size, max_size;

public:
  // Constructeurs / Destructeurs
  Tableau();
  Tableau(unsigned int);
  Tableau(Tableau const&);
  ~Tableau();

  // Iterateurs
  int get(unsigned int);
  int operator[](unsigned int i);

  // Selecteurs
  unsigned int size();

  // Modificateurs
  void resize(unsigned int);
  void set(unsigned int, int);
};
```

- Et maintenant, à votre avis, comment faire un tableau de `double` ?
  - duplication de code

- Et maintenant, à votre avis, comment faire un tableau de `double` ?
  - duplication de code
- ... ensuite, à votre avis, comment faire un tableau de `float` ?
  - encore de la duplication de code

- Et maintenant, à votre avis, comment faire un tableau de `double` ?
  - duplication de code
- ... ensuite, à votre avis, comment faire un tableau de `float` ?
  - encore de la duplication de code
- ... encore mieux, à votre avis, comment faire un tableau d'un type quelconque ?
  - encore et encore de la duplication de code

- Et maintenant, à votre avis, comment faire un tableau de `double` ?
  - duplication de code
- ... ensuite, à votre avis, comment faire un tableau de `float` ?
  - encore de la duplication de code
- ... encore mieux, à votre avis, comment faire un tableau d'un type quelconque ?
  - encore et encore de la duplication de code

$\rightsquigarrow$ On utilise les templates !

Plan du cours

1. Structure de données abstraites

2. Vers la généricité de ces structures
   - Rappel de ce qu'est un template
   - Modèle de fonctions/méthodes
   - Modèle de classes

Qu'est ce que un template ?

- Un template ou patron/modèle $\rightsquigarrow$ modèle à partir duquel des fonctions/méthodes ou des classes pourront être générées automatiquement par le compilateur en fonction d'une série de paramètres : c'est le principe de généricité
  - par exemple : création d'une classe paramétrée par le type d'un de ses attributs

- Remarque : c'est un niveau d'abstraction supplémentaire

Qu'est ce que un template ?

- Un template ou patron/modèle $\rightsquigarrow$ modèle à partir duquel des fonctions/méthodes ou des classes pourront être générées automatiquement par le compilateur en fonction d'une série de paramètres : c'est le principe de généricité
  - par exemple : création d'une classe paramétrée par le type d'un de ses attributs
- Remarque : c'est un niveau d'abstraction supplémentaire
- À chaque fois qu'un template est utilisé avec une série de paramètres différents, une nouvelle version de la classe ou de la fonction/méthode est créée
  - on parle d'instanciation du template
  - et spécialisation du template

Qu'est ce que un template ?

- Un template ou patron/modèle $\rightsquigarrow$ modèle à partir duquel des fonctions/méthodes ou des classes pourront être générées automatiquement par le compilateur en fonction d'une série de paramètres : c'est le principe de généricité
  - par exemple : création d'une classe paramétrée par le type d'un de ses attributs
- Remarque : c'est un niveau d'abstraction supplémentaire
- À chaque fois qu'un template est utilisé avec une série de paramètres différents, une nouvelle version de la classe ou de la fonction/méthode est créée
  - on parle d'instanciation du template
  - et spécialisation du template
- En C++, il n'y a pas d'opérateur ou de mot clé réservé pour instancier un modèle
  - instanciation à l'utilisation (cf. suite du cours)

Comment définir un template ?

- Définition d'un template : une liste de paramètres $\rightsquigarrow$ des types ou des valeurs
  - une valeur est indiquée par son type : `int`, `double`, ...
  - un type est indiqué par le mot réservé `class`

  le tout précédé du mot clé `template`

```
template<class T, int N>
```

Comment définir un template ?

- Définition d'un template : une liste de paramètres $\rightsquigarrow$ des types ou des valeurs
  - une valeur est indiquée par son type : `int`, `double`, ...
  - un type est indiqué par le mot réservé `class`

  le tout précédé du mot clé `template`

```
template<class T, int N>
```

- Remarque : ici, le mot clé `class` ne signifie pas que `T` est une classe mais uniquement un type $\rightsquigarrow$ pour lever l'ambiguïté, depuis la norme ISO, on peut utiliser le mot clé `typename` au lieu de `class`

Comment définir un template ?

- Définition d'un template : une liste de paramètres $\rightsquigarrow$ des types ou des valeurs
  - une valeur est indiquée par son type : `int`, `double`, ...
  - un type est indiqué par le mot réservé `class`

  le tout précédé du mot clé `template`

```
template<class T, int N>
```

- Remarque : ici, le mot clé `class` ne signifie pas que `T` est une classe mais uniquement un type $\rightsquigarrow$ pour lever l'ambiguïté, depuis la norme ISO, on peut utiliser le mot clé `typename` au lieu de `class`

```
template<class T>
T min(T a, T b)
{
  return (a < b ? a : b);
}
```

```
template<typename T>
T min(T a, T b)
{
  return (a < b ? a : b);
}
```

Définition et instanciation de modèles de fonctions

- L'instanciation d'un modèle de fonction/méthode est commandée explicitement lors de l'appel d'une des fonctions que le modèle définit

```
template<typename T>
T min(T tab[], int n) // minimum d'un tableau de n elements de type T
{                     // -> les elements de type T doivent pouvoir etre
                      //    compares
  T min = tab[0]; for (int i = 1; i < n; i++)
    if (tab[i] < min) min = tab[i];
  return min;
}

int
main(void)
{
  int t[] = { 10, 5, 8, 14, 20, 3, 19, 7 };
  string c = "BKEFYFFLKRNFAJDQKXJD";

  cout << min<int>(t, 8) << endl;                          // 3 -> T = int
  cout << min<char>((char*)c.c_str(), c.size()) << endl;   // A -> T = char
  return 0;
}
```

Définition et instanciation de modèles de fonctions

- L'instanciation d'un modèle de fonction/méthode est commandée explicitement lors de l'appel d'une des fonctions que le modèle définit

```
template<typename T>
T min(T tab[], int n) // minimum d'un tableau de n elements de type T
{                     // -> les elements de type T doivent pouvoir etre
                      //    compares
  T min = tab[0]; for (int i = 1; i < n; i++)
    if (tab[i] < min) min = tab[i];
  return min;
}

int
main(void)
{
  int t[] = { 10, 5, 8, 14, 20, 3, 19, 7 };
  string c = "BKEFYFFLKRNFAJDQKXJD";

  cout << min(t, 8) << endl;                          // 3 -> T = int
  cout << min((char*)c.c_str(), c.size()) << endl;    // A -> T = char
  return 0;
}
```

- Remarque : les arguments du modèle peuvent être tous omis si leur valeur peut être déterminée

Fonction ordinaire vs. fonction patron

- Lorsque les instances des fonctions patrons (surcharge de fonctions) sont en concurrences avec des fonctions ordinaires :
  - les fonctions ordinaires sont préférées aux fonctions patrons
  - les fonctions patrons les plus spécialisées sont préférées à celles qui le sont moins

```
template<typename T>
void foo(T a, T b){ cout << "Fonction foo (1)" << endl; }

template<typename T>
void foo(T a, const char b[]){ cout << "Fonction foo (2)" << endl; }

void foo(const char a[], const char b[]){ cout << "Fonction foo (3)" << endl; }

int
main(void)
{
  foo(1.0,2.0);    // On est dans la fonction foo (1)
  foo(1.0,"2");    // On est dans la fonction foo (2)
  foo("1","2");    // On est dans la fonction foo (3)
  return 0;
}
```

Spécialisation explicite de fonctions patrons

- Spécialisation explicite d'une fonction patron $\rightsquigarrow$ définit un modèle spécifique de fonction à utiliser pour un jeu de paramètres donné

```
template<typename T>
void
print(T arg)
{
  cout << "Impossible determiner le type T" << endl;
}

template<>
void
print<int>(int arg)
{
  cout << "Determiner le type T -> int" << endl;
}

int
main(void)
{
  print<char>('a');    // Impossible determiner le type T
  print<int>(17);      // Determiner le type T -> int

  return 0;
}
```

Définition et instanciation de modèles de classes

- La définition d'un modèle de classe se fait de manière similaire à le définition d'un modèle de fonction

```
template <typename T>
class Intervalle
{
  T binf, bsup;       // declaration de deux objets de type T

public:
  Intervalle();
  Intervalle(T, T);
  void print();       // ...
};
```

Définition et instanciation de modèles de classes

- La définition d'un modèle de classe se fait de manière similaire à le définition d'un modèle de fonction

```
template <typename T>
class Intervalle
{
  T binf, bsup;       // declaration de deux objets de type T

public:
  Intervalle();
  Intervalle(T, T);
  void print();       // ...
};
```

- L'instanciation d'un modèle de classe se fait à la déclaration d'un objet du type de la classe

```
int
main(void)
{
  Intervalle<int> i1(0,1); // ...
  return 0;
}
```

Déclaration d'une méthode à l'extérieure de la classe

- La déclaration d'une méthode (méthode ordinaire, constructeur, ou destructeur) se fait de manière similaire à celle d'une classe ordinaire
  - mais elles doivent également comporter des paramètres

```
#include "Intervalle.hpp"

template<typename T>
Intervalle<T>::Intervalle():binf(0),bsup(0)
{
  /* ... */
}

template<typename T>
Intervalle<T>::Intervalle(T binf , T bsup):binf(binf),bsup(bsup)
{
  /* ... */
}

template<typename T>
void Intervalle<T>::print()
{
  /* ... */
}
```

Spécialisation explicite de classes patrons

- Spécialisation explicite d'une classe patron $\rightsquigarrow$ de la même manière que pour les fonctions patrons, définit un modèle spécifique de classe à utiliser pour un jeu de paramètres donné

```
template<typename T>
class Test
{
public:
  void print(){ cout << "Class Test<T>" << endl; }
};

template<>
class Test<double>
{
public:
  void print(){ cout << "Class Test<double>" << endl; }
};
```

```
int
main(void)
{
  Test<int> tmp1; tmp1.print();
  Test<double> tmp2; tmp2.print();
  return 0;
}
```

Paramètre avec valeur par défaut pour les classes patrons

- Contrairement aux fonctions patrons, les paramètres des classes patrons peuvent avoir une valeur par défaut, sauf en cas de spécialisation explicite :
  - affectation d'un type par défaut aux paramètres de type,
  - et affectation d'une valeur par défaut aux paramètres non type.

```
template<typename T, int Max = 17>
class Tableau
{
private:
  T* elts;
public:
  Tableau(int n){
    cout << "Creation d'un tableau de " << min(n,Max) << " elements." << endl;
    elts = new T[min(n,Max)];
  } // ...
};
```

Paramètre avec valeur par défaut pour les classes patrons

- Contrairement aux fonctions patrons, les paramètres des classes patrons peuvent avoir une valeur par défaut, sauf en cas de spécialisation explicite :
  - affectation d'un type par défaut aux paramètres de type,
  - et affectation d'une valeur par défaut aux paramètres non type.

```
template<typename T, int Max = 17>
class Tableau
{
private:
  T* elts;
public:
  Tableau(int n){
    cout << "Creation d'un tableau de " << min(n,Max) << " elements." << endl;
    elts = new T[min(n,Max)];
  } // ...
};
```

```
main(void)
{
  Tableau<int,20> tmp1(40);      // Creation d'un tableau de 20 elements.
  Tableau<int> tmp2(40);         // Creation d'un tableau de 17 elements.
  return 0;
}
```

Définition de la classe patron Tableau

- Définition d'un tableau d'objets de type `Type`

```
template<typename Type>
class Tableau
{
private:
  Type * elts;
  unsigned int current_size, max_size;

public:
  // Constructeurs / Destructeurs
  Tableau();
  Tableau(unsigned int);
  Tableau(Tableau<Type> const&);
  ~Tableau();

  // Iterateurs
  Type get(unsigned int);
  Type operator[](unsigned int i);

  // Selecteurs
  unsigned int size();

  // Modificateurs
  void resize(unsigned int);
  void set(unsigned int, Type);
};
```

À vos pinceaux !

Exercice (Définition de la classe paramétrée Tableau)

En vous inspirant de la déclaration de la classe Tableau du transparent précédent, reprendre l'Exercice 1 du TD2 et écrire la classe paramétrée Tableau.

Questions ?