Programmation Orientée Objet : Templates : les classes et fonctions génériques

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs du cours d'aujourd'hui

Le but de ce cours est de présenter les **modèles de classes et de fonctions**, basés sur le concept de programmation générique (« generic programming »).

On parle aussi pour cela de **polymorphisme paramétrique**.

Note : Ce cours est un **cours avancé** dont le contenu ne fera pas partie des examens.

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Un exemple

Prenons un exemple simple pour commencer :
une fonction échangeant la valeur de 2 variables.

Par exemple avec 2 entiers vous écririez une fonction comme :

```
// échange la valeur de ses arguments
void echange(int& i, int& j) {
  int tmp(i);
  i = j;
  j = tmp;
}
```

Mais vous vous rendez bien compte que vous pourriez faire la même chose (le **même algorithme**) avec deux `double`, ou même deux objets quelconques, pour peu qu'ils aient un constructeur de copie (`Obj tmp(i);`) et un opérateur de copie (`operator=`).

Jean-Cédric Chappelier
& Jamila Sam

Exemple, suite...

L'écriture générale serait alors quelque chose comme :

```
// échange la valeur de ses arguments
void echange(Type& i, Type& j) {
  Type tmp(i);
  i = j;
  j = tmp;
}
```

où `Type` est une représentation **générique** du type des objets à échanger.

La façon exacte de le faire en C++ est la suivante :

```
// échange la valeur de ses arguments
template<typename Type>
void echange(Type& i, Type& j) {
  Type tmp(i);
  i = j;
  j = tmp;
}
```


Jean-Cédric Chappelier
& Jamila Sam
EPFL

...et fin

On pourra alors utiliser la fonction `echange` avec tout type/classe pour lequel le constructeur de copie et l'opérateur d'affectation (`=`) sont définis.

Par exemple :

```
int a(2), b(4);
echange(a, b);

double da(2.3), db(4.5);
echange(da, db);

vector<double> va, vb;
...
echange(va, vb);

string sa("ca marche"), sb("coucou");
echange(sa, sb);
```

Note : Un tel *modèle de fonctions* existe dans la bibliothèque `algorithm` et s'appelle `swap`.

Jean-Cédric Chappelier
& Jamila Sam

Programmation générique

L'idée de base est de **passer les** types de données **comme paramètres**
pour décrire des traitements très généraux (« génériques »)

Il s'agit donc d'un *niveau d'abstraction supplémentaire*.

De tels modèles de classes/fonctions s'appellent aussi classes/fonctions génériques ou patrons (chablons), ou encore « ***template*** ».

Vous en connaissez déjà sans le savoir.
Par exemple la « classe » `vector` n'est en fait pas une classe, mais un *modèle de classes* :
c'est le même modèle que l'on stocke des `char` (`vector<char>`),
des `int` (`vector<int>`), ou tout autre objet (p.ex. `vector<Figure*>`).

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Modèle de classes

Ce que l'on a fait dans l'exemple introductif avec des fonctions (`echange()`) peut aussi être fait avec des classes.

On pourrait par exemple vouloir créer un modèle de classes qui réalise une paire d'objets quelconques :

```
template<typename T1, typename T2>
class Paire {
public:
    Paire(const T1& un, const T2& deux)
      : premier(un), second(deux) {}
    T1 get1() const { return premier; }
    T2 get2() const { return second;  }
    void set1(const T1& val) { premier = val; }
    void set2(const T2& val) { second  = val; }
protected:
    T1 premier;
    T2 second;
};
```


Jean-Cédric Chappelier
& Jamila Sam

Généralisation aux classes (2)

et par exemple créer la classe « paire `string`–`double` » :

```
Paire<string,double>
```

ou encore la classe « paire `char`–`unsigned int` » :

```
Paire<char,unsigned int>
```

Note : un tel modèle de classes existe dans la bibliothèque `utility` et s'appelle `pair`.

Les modèles de classes sont donc **un moyen condensé d'écrire plein de classes potentielles à la fois**.

(de même que les modèles de fonctions/méthodes sont un moyen condensé d'écrire **plein** de fonctions/méthodes potentielles à la fois)

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Déclaration d'un modèle

Pour déclarer un modèle de classes ou de fonctions, il suffit de faire précéder sa déclaration du mot clé `template` suivi de ses paramètres (qui sont donc des noms génériques de type) suivant la syntaxe :

`template<typename nom1, typename nom2, ...>`

Exemple :

```
template<typename T1, typename T2>
class Paire {
  ...
```

Les types ainsi déclarés (paramètres du modèle) peuvent alors être utilisés dans la définition qui suit, exactement comme tout autre type.

Note : on peut aussi utiliser le mot `class` à la place de `typename`, par exemple :

```
template<class T1, class T2>
class Paire {
  ...
```


Jean-Cédric Chappelier
& Jamila Sam

Déclaration d'un modèle (2)

Il est également possible de définir des types par défaut, avec la même contrainte que pour les paramètres de fonction : les valeurs par défaut doivent être placées en dernier.

Exemple :

```
template<typename T1, typename T2 = unsigned int>
class Paire {
  ...
```

qui permettrait de déclarer la classe « paire `char`–`unsigned int` » simplement par :

`Paire<char>`

On peut même faire

```
template<typename T1, typename T2 = T1>
class Paire {
  ...
```


Jean-Cédric Chappelier
& Jamila Sam

Définitions externes des méthodes de modèles de classes

Si les méthodes d'un modèle de classes sont définies en dehors de cette classe, elle devront alors aussi être définies comme modèle et être précédées du mot clé `template`, mais...

...il est **de plus absolument nécessaire** d'**ajouter les paramètres du modèle** (les types génériques) **au nom de la classe**
[pour bien spécifier que dans cette définition c'est la classe qui est en modèle et non la méthode.]

Exemple :

```
template<typename T1, typename T2>
class Paire {
public:
  Paire(const T1&, const T2&);
  ...
};

// définition du constructeur
template<typename T1, typename T2>
Paire<T1,T2>::Paire(const T1& un, const T2& deux)
  : premier(un), second(deux) { }
```

Instanciation des modèles

La définition des modèles ne génère en elle-même aucun code : c'est juste une **description de plein de codes potentiels**.

Le code n'est produit que lorsque tous les paramètres du modèle ont pris chacun un type spécifique.

Lors de l'utilisation d'un modèle, il faut donc fournir des valeurs pour tous les paramètres (au moins ceux qui n'ont pas de valeur par défaut).
On appelle cette opération une **instanciation** du modèle.

L'instanciation peut être implicite lorsque le contexte permet au compilateur de décider de l'instance de modèle à choisir.

Par exemple, dans le code :

```
double da(2.3), db(4.5);
echange(da, db);
```

il est clair (par le contexte) qu'il s'agit de l'instance `echange<double>` du modèle `template<typename T> void echange(T&,T&);` qu'il faut utiliser.

Jean-Cédric Chappelier & Jamila Sam

Instanciation des modèles (2)

Mais dans la plupart des cas, on **explicite l'instanciation** lors de la déclaration d'un objet.
C'est ce qui vous faites lorsque vous déclarez par exemple `vector<double> tableau;`

Il suffit dans ce cas de spécifier le(s) type(s) désiré(s) après le nom du modèle de classes et entre `<>`.

L'instanciation explicite peut *aussi* être utile dans les cas où le contexte n'est pas suffisamment clair pour choisir.

Par exemple avec le modèle de fonctions

```
template <typename Type>
Type monmax(const Type& x, const Type& y)
{
  if (x < y) return y;
  else       return x;
}
```

l'appel « `monmax(3.14, 7);` » est ambigu.

Il faudra alors écrire :
« `monmax<double>(3.14, 7);` »

Note C++17 : on peut aussi faire de l'instanciation implicite de modèle de classes :

```
vector tab(5, 1.0);
```


Jean-Cédric Chappelier
& Jamila Sam

Modèles, surcharge et spécialisation

Les modèles de fonctions peuvent très bien être surchargés comme les fonctions usuelles (puisque, encore une fois, ce sont juste des façons condensées d'écrire plein de fonctions à la fois).

Par exemple :

```
template<typename Type>
void affiche(const Type& t) {
  cout << "J'affiche " << t << endl;
}

// surcharge pour les pointeurs : on préfère ici écrire
//   le contenu plutôt que l'adresse.
template<typename Type>
void affiche(Type* t) {
  cout << "J'affiche " << *t << endl;
}
```

Note : on aurait même pu faire mieux en faisant appel au premier modèle :

```
template<typename Type>
void affiche(Type* t) { affiche<Type>(*t); }
```


Jean-Cédric Chappelier & Jamila Sam

Modèles, surcharge et spécialisation (2)

Mais les modèles (y compris les modèles de classes) offrent un mécanisme supplémentaire : la **spécialisation** qui permet de définir une version particulière d'une classe ou d'une fonction pour un choix spécifique des paramètres du modèle.

Par exemple, on pourrait spécialiser le second modèle ci-dessus dans le cas des pointeurs sur des entiers :

```
template<> void affiche<int>(int* t) {
  cout << "J'affiche le contenu d'un entier : " << *t
       << endl;
}
```

La spécialisation d'un modèle (lorsqu'elle est totale) se fait en :

- ajoutant `template<>` devant la définition
- nommant explicitement la classe/fonction spécifiée
  C'est le `<int>` après `affiche` dans l'exemple ci-dessus.

Jean-Cédric Chappelier
& Jamila Sam

Exemple de spécialisation de classe

```
template<typename T1, typename T2>
class Paire {
...
};

// specialisation pour les paires <string,int>
template<> class Paire<string,int> {
public:
  Paire(const string& un, int deux)
    : premier(un), second(deux) {}
  string get1() const { return premier; }
  int get2()    const { return second;  }
  void set1(const string& val) { premier = val; }
  void set2(int val) { second  = val; }

  // une methode de plus
  void add(int i) { second += i; }

protected:
  string premier;
  int    second;
};
```

(Question : comment aurait-on pu faire mieux ?)

Jean-Cédric Chappelier & Jamila Sam

Spécialisation : remarques

Note 1 : La spécialisation peut également s'appliquer uniquement à une méthode d'un modèle de classes sans que l'on soit obligé de spécialiser toute la classe. Utilisée de la sorte, la spécialisation peut s'avérer particulièrement utile.

Note 2 : La spécialisation n'est pas une surcharge car il n'y a pas génération de plusieurs fonctions de même nom (de plus que signifie une surcharge dans le cas d'une classe ?) mais bien une **instance spécifique** du modèle.

Note 3 : Pour les **classes** (pas pour les fonctions), il existe aussi des spécialisations partielles (de toute la classe ou de certaines méthodes), mais cela nous emmènerait trop loin dans ce cours.

Note 4 : Pour les **fonctions**, préférez la surcharge à la spécialisation :

```
void affiche(int* t) {
  cout << "J'affiche le contenu d'un entier : " << *t
       << endl;
}
```

Modèles paramétrés par des entiers

En plus des modèles paramétrés par un *type*, on peut aussi écrire des modèles **paramétrés par des entiers**, des valeurs enumérées, des pointeurs ou des références.

Illustrons ici sur des modèles paramétrés par des entiers :

Exemple 1 : `array<double, 3>`

Exemple 2 : une classe représentant (le concept d')un intervalle d'entiers : entre N inclu et M exclu : $[N, M[$

```
template <int N, int M>
class IntRange
{
public:
  int low()  const { return min(N,M); }
  int high() const { return max(N,M); }

  bool contains(int x) const {
    return (x >= low()) and (x < high());
  }
};
```


Jean-Cédric Chappelier & Jamila Sam

suite de l'exemple

On pourrait même l'afficher :

```
template <int N, int M>
ostream& operator<<(ostream& flot, IntRange<N, M> const& r) {
  return flot << '[' << r.low() << ", " << r.high() << '[';
}
```

et créer un itérateur sur cette classe (voir semaine prochaine) :

```
class IntRangeIterator {
public:
  IntRangeIterator(int index) : index_(index) {}

  bool operator==(const IntRangeIterator& x) const
  { return index_ == x.index_;  }

  bool operator!=(const IntRangeIterator& x) const
  { return not(*this == x);  }

  int operator*() const  { return index_; }
  IntRangeIterator& operator++() { ++index_; return *this; }

private:   int index_; // se souvient où l'on est
};
```


Jean-Cédric Chappelier & Jamila Sam

suite de l'exemple (2)

```
template <int N, int M>
class IntRange
{
public:
  typedef IntRangeIterator const_iterator;

  const_iterator begin() const
  { return const_iterator(low()); }

  const_iterator end() const
  { return const_iterator(high()); }
...
};

template <typename T>
void show_all(const T& t)
{
  for (auto i : t) cout << i << ", ";
  cout << "[END]" << endl;
}
...
  show_all( IntRange<10, 20>() );
...
```


Jean-Cédric Chappelier
& Jamila Sam
EPFL

Modèles paramétrés par des entiers (2)

On peut bien sûr aussi :

- spécialiser ces modèles paramétrés par des entiers

```
template <int N>
class Factorial {
    enum { value = N * Factorial<N - 1>::value };
};

template <>
class Factorial<0> {
    enum { value = 1 };
};
```

- combiner des paramètres de types et entiers :

```
template <typename type, int N>
class MachinTruc {
...
protected:
  array<type, N> contenu;
}
```


Jean-Cédric Chappelier & Jamila Sam

Modèles de classes et compilation séparée

Les modèles de classes doivent nécessairement être définis au moment de leur instanciation afin que le compilateur puisse générer le code correspondant.

Ce qui implique, lors de compilation séparée, que les fichiers d'en-tête (`.h`) doivent contenir non seulement la déclaration, mais également la définition complète de ces modèles !!

On ne peut donc pas séparer la déclaration de la définition dans différents fichiers...
Ce qui présente plusieurs inconvénients :

- les mêmes instances de modèles peuvent être compilées plusieurs fois ;
- et peuvent se retrouver en de multiples exemplaires dans les fichiers exécutables ;
- on ne peut plus cacher leurs définitions (p.ex. pour des raisons de confidentialité, protection contre la concurrence, etc.).

Jean-Cédric Chappelier
& Jamila Sam

Templates

Déclarer un modèle de classe ou de fonction :

```
template<typename nom1, typename nom2, ...>
```

Définition externe des méthodes de modèles de classes : `template<typename nom1, typename nom2, ...> NomClasse<nom1, nom2, ...>::NomMethode(...`

Instanciation : spécifier simplement les types voulus après le nom de la classe/fonction, entre <> (Exemple : `vector<double>`)

Spécialisation (totale) de modèle pour les types *type1*, *type2*... :

```
template<>
NomModele<type1,type2,...>
...suite de la declaration...
```

Compilation séparée : pour les templates, il faut tout mettre (déclarations et définitions) dans le fichier d'en-tête (`.h`).

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Ce que j'ai appris aujourd'hui

À pouvoir faire des modèles génériques de fonctions ou de classes, indépendamment du type, ce que l'on appelle de la **programmation générique**.

Nous avons vu :

- comment déclarer de tels modèles
- comment en créer des instances
- comment spécialiser certains modèles