ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

2e année – Spécialité Informatique

Année 2024 – 2025

Programmation orientée objet en C++

Notes de cours

Sébastien Fourey

Version du 3 septembre 2024

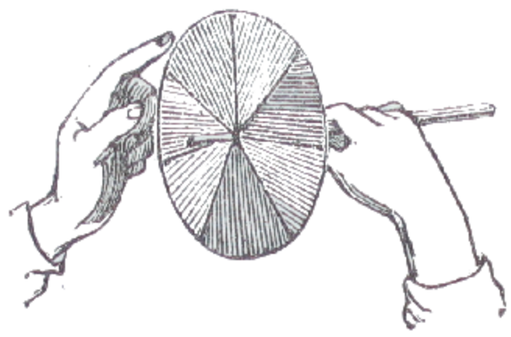

Fig. 46. — La lumière *blanche* est produite par la *réunion* des sept couleurs du spectre.

Chapitre 3

Programmation générique grâce aux modèles

3.1 Motivation

Il est très fréquent que des algorithmes complets ne diffèrent que par le type de certaines des données qu'ils manipulent. En terme d'implémentation dans un langage particulier, une fonction entière peut être applicable à des types de données différents, et ce à l'opérateur près. Il peut aussi arriver qu'en dehors des types utilisés, seulement quelques-unes des opérations réalisées soient dépendantes des types manipulés. Il semble alors justifié de vouloir réduire la quantité de code à écrire quand on souhaite utiliser ce genre d'algorithmes pour plusieurs types de données.

Le polymorphisme tel qu'il a été présenté dans le chapitre 2 peut s'avérer très utile pour répondre à ce besoin. Pensez par exemple à un algorithme qui ne manipule que des figures. Un tel algorithme, tant qu'il n'utilise que des opérations propres aux figures (et pas à ses types dérivés), peut en fait s'appliquer indifféremment à des cercles, rectangles ou autres.

D'autre part, ce qui est valable pour une fonction l'est aussi pour un ensemble de fonctions et donc pour une classe. Par exemple, quelle serait la différence entre la classe des ensembles d'entiers du listing 3.1 et une classe d'ensembles de nombres flottant ?

Toutefois, il y a des situations pour lesquelles la solution apportée par le polymorphisme n'est pas satisfaisante, entre autres pour les quelques raisons qui suivent :

— Le polymorphisme, appelé dans ce cas *polymorphisme d'exécution*, constitue une solution *dynamique* à un problème qui peut parfois être résolu dès la compilation.
— Si tel est le cas, on peut juger que « l'artillerie » des méthodes virtuelles, avec les indirections et les contrôles dynamiques qu'elle implique, entraîne une perte d'efficacité dont on pourrait se passer.
— L'utilisation de l'héritage, comme moyen de regrouper au sein d'une même famille les classes d'objets qui *doivent pouvoir* être utilisées par un algorithme donné, peut devenir rapidement très arbitraire et contre-intuitif.
— Finalement, la lourdeur introduite est flagrante lorsqu'il s'agit de manipuler les types de base du langage. (Pensez à une classe *Integer*, comme en Java, et à un opérateur d'addition pour les types numériques qui serait une méthode virtuelle !)

Finalement, un programmeur (p. ex. en C) aura simplement pris l'habitude dans certains cas de dupliquer son code source pour produire des versions adaptées par de simples substitutions automatiques effectuées à l'aide de son éditeur favori. En première approximation, on peut dire que le C++ offre alors une solution analogue mais qui fait partie intégrante du langage : les modèles, sans doute aussi connus sous le nom de *templates*.

```
1  class SetOfIntegers {
2    int * array;
3    unsigned int cardinality;
4    unsigned int capacity;
5
6  public:
7    void SetOfIntegers(int capacity = 0);
8    void add(int);
9    bool contains(int);
10   // ...
11 };
```

Listing 3.1 – Classe des ensembles d'entiers.

3.2 Modèle de classe

Un *modèle de classe* (ou *patron de classe*) est une définition de classe paramétrée par un ou plusieurs types. La définition d'une classe pour des types précis, à l'aide d'un modèle, porte le nom d'*instanciation de modèle*.

Avant de donner la syntaxe C++ utilisée, le diagramme de la figure 3.1 montre la notation UML correspondant aux modèles de classes.

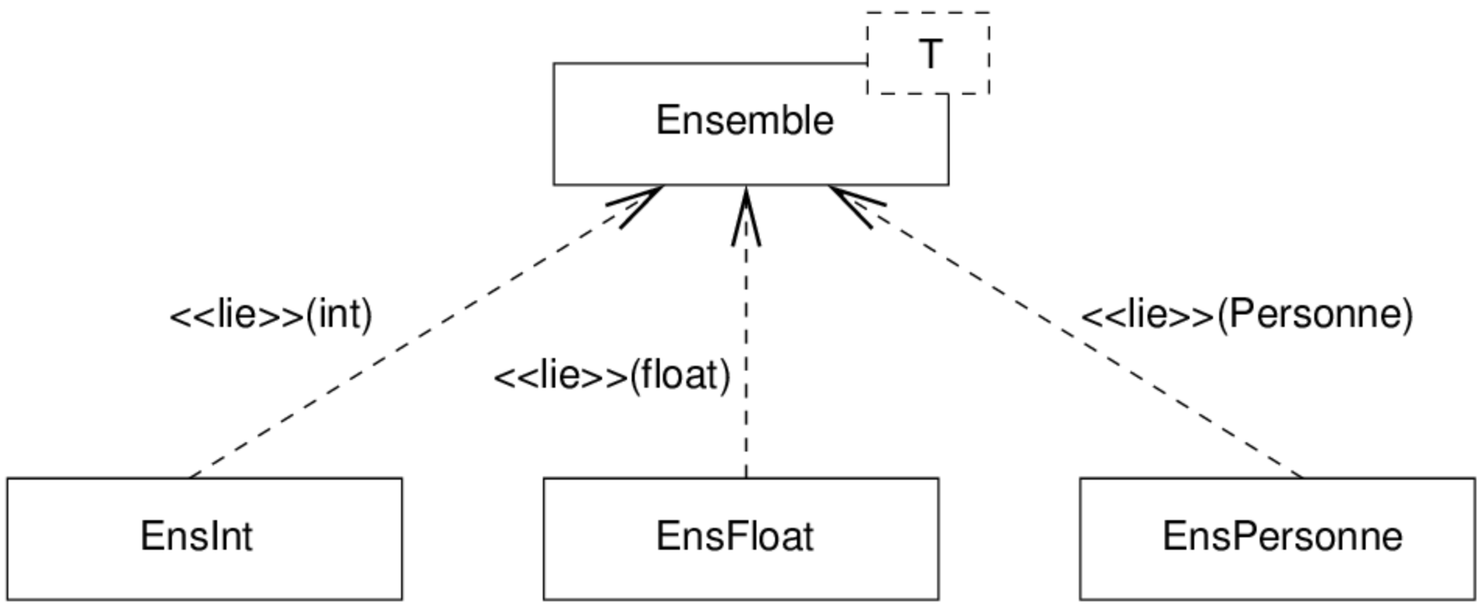

FIGURE 3.1 – Le modèle de classe Ensemble et trois de ses instanciations possibles.

3.2.1 Déclaration

Un modèle de classe, paramétré par un ou plusieurs types, est déclaré de l'une des manières suivantes :

```
template<class T> class NomMod {
  // Définition utilisant T comme un type classique.
```

```
};
```

```
template<class T, class U,... > class NomMod {
  // Définition utilisant T, U, ... comme des types classiques.
};
```

```
template<typename T> class NomMod {
  // Définition utilisant T comme un type classique.
};
```

```
template<typename T, class U,... > class NomMod {
  // Définition utilisant T, U, ... comme des types classiques.
};
```

D'un point de vue syntaxique, les mots-clés `typename` et `class` qui précèdent chaque paramètre du modèle sont équivalents. Le mot-clé `typename` peut être jugé plus lisible car il ne laisse pas supposer que le paramètre devrait être un nom de classe. En effet, un type de base ou tout autre type défini par l'utilisateur peut être utilisé pour instancier un modèle.

En fait, un paramètre de modèle peut aussi être une valeur constante d'un certain type, et pas uniquement un nom de type, comme le montre le listing 3.2. Les types autorisés sont cependant limités :

- types intégrals (int, char, etc.) ou énumérés ;
- pointeurs ;
- références.

Notamment, il n'est pas possible d'utiliser comme argument de template une constante de type float. De plus, dans le cas d'un argument de type pointeur (ou référence), l'adresse en question doit être connue à la compilation. Il ne peut donc pas s'agir d'une variable locale !

```
1  #include <iostream>
2
3  template <typename T, int size>
4  class Array {
5    T cells[size];
6
7  public:
8    Array();
9    const T & operator[](int) const;
10   T & operator[](int);
11 };
12
13 int main() {
14   int s;
15   Array<int, 100> array; // OK
16   std::cout << "Enter a size: " << std::flush;
17   std::cin >> s;
18   Array<int, s> dynamicArray; // Error!
19 }
```

Listing 3.2 – Utilisation d'une constante comme paramètre de type.

De plus, un paramètre de modèle peut avoir un *type* ou une *valeur* par défaut, comme illustré dans le listing 3.3.

```cpp
#include <iostream>
template <typename T, int columns, int rows = 1>
class Matrix {
  T cells[rows][columns];

public:
  Matrix();
  const T & operator()(int, int) const;
  T & operator()(int, int);
};

int main() {
  int s;
  Matrix<double, 100, 100> matrixA; // Matrix 100 by 100
  Matrix<double, 100> matrixX;      // Row vector
}
```

Listing 3.3 – Modèle de classe dont un paramètre possède une valeur par défaut.

Enfin, point important, un modèle peut hériter d'autres modèles. Comme on le verra dans la section 3.3, deux instances d'un même modèle étant en fait deux classes bien distinctes, il est possible que la spécialisation d'un modèle hérite d'une instanciation du modèle en question.

3.2.2 Définition d'une méthode hors de la déclaration du modèle

La définition d'une méthode inline à l'intérieur de la définition de la classe est immédiate, mais le mot-clé *template* doit être réutilisé lors d'une définition à l'extérieur à la classe.

```
template<typename T> class NomModèle {
  [...]
   typeRetour idMéthode(T arg1,...); // Simple déclaration
};

template<typename T> typeRetour NomModèle<T>::idMéthode(T arg1,...) {
  // Code de la méthode
}
```

D'une manière générale, on peut se rappeler que le nom d'un modèle de classe ne peut apparaître qu'affublé des symboles < >. L'instanciation d'un modèle dont tous les paramètres ont une valeur par défaut ne fait pas exception à la règle [...]. La mise en garde suivante va dans le même sens.

Un modèle de classe ne peut pas être surchargé par une définition de classe. (Mais des spécialisations sans paramètres sont possibles.)

3.2.3 Instanciation d'un modèle : vérifications et variantes

Comme cela a été présenté en introduction, l'instanciation d'un modèle résulte en la définition d'une classe C++ habituelle, et donc la génération du code correspondant. En effet, un modèle

présent dans un fichier source mais qui n'est pas instancié ne produit aucun code. La phase de vérification syntaxique et sémantique est plus que succincte pour un modèle tant qu'il n'est pas instancié. Ce n'est que lorsque tous les types sont connus et que le code d'une classe est généré que ce dernier peut être vérifié. Cette phase, l'instanciation, est provoquée dès que le nom du modèle apparaît avec des paramètres totalement renseignés. Elle peut prendre plusieurs formes, qui sont énumérées ici.

Quand un modèle est utilisé pour des types particuliers dans un programme, il faut s'assurer que le code correspondant *peut* être généré (c.-à-d., le compilateur dispose de la définition complète du modèle) ou bien qu'il a été généré dans une autre unité de compilation et sera présent lors de l'édition de liens. Ceci correspond à des niveaux de généricité bien distincts [...].

Première méthode La méthode d'instanciation de modèle la plus concise passe par la déclaration d'un objet (y compris comme paramètre d'une fonction) :

NomModèle<*type*,...> nom_objet ;

NomModèle<*type*,...> nom_objet(arg1, ... , argN) ;

Deuxième méthode Le mot-clé `template` peut aussi être utilisé pour déclencher l'instanciation, mais en ne définissant cette fois ci aucun objet. Cette méthode est préférée dans le cas d'une bibliothèque pour laquelle il serait dommage de devoir créer des variables (globales) inutilisées.

`template class` NomModèle<*type*,...>;

Ces différentes méthodes d'instanciation d'un modèle de classe sont illustrées par le code suivant :

```
1 List<int> primes;
2 class Student {
3   // ...
4 };
5 List<Student> my_group;
6 template class List<Student>;
```

Listing 3.4 – Différents types d'instanciation d'un modèle de classe.

Exercice 3.1 (Question d'examen) *Écrivez un code (très court) définissant un modèle de classe rudimentaire. Ensuite, donnez une ligne de code réalisant l'instanciation de votre modèle sans aucune déclaration ni construction de variable.*

3.2.4 C++11 : Empêcher l'instanciation

En C++11, il est possible d'empêcher l'instanciation d'un modèle dans une unité de compilation. En effet, l'instanciation à plusieurs reprises d'un même modèle dans différentes unités de compilation pose un problème d'efficacité (de la compilation) non négligeable. On peut donc empêcher cette instanciation grâce au mot-clé `extern` comme dans l'exemple du listing 3.5.

```
1 extern template class List<Student>;
```

Listing 3.5 – Instruction `extern` pour un template.

3.2.5 Exemple

Le listing 3.6 montre le « squelette » d'un modèle de classe complet qui a la particularité, néanmoins classique, de posséder un modèle de fonction amie. La syntaxe, mais aussi l'ordre des déclarations mérite que le lecteur s'y attarde. En effet, pareille définition d'un modèle de fonction amie se fait en pas moins de 4 étapes.

3.3 Spécialisations définies par l'utilisateur

Motivation

Les modèles permettent d'écrire des algorithmes génériques avec une syntaxe intégrée au langage. Mais il arrive parfois que cette généricité aille à l'encontre de l'efficacité. En effet, certains types peuvent posséder des particularités qui, si on les utilise, permettent d'accélérer sensiblement l'exécution d'un programme.

Il peut aussi arriver, plus simplement, qu'une opération utilisée dans un modèle n'existe pas pour une type donné. Dans ce cas comme dans le précédent, on peut écrire une ou plusieurs variantes du modèle adaptées à des types précis. On parle alors de *spécialisation(s)* du modèle.

```
1 Vector<int> v;
2 Vector<Shape> vs;
3 Vector<Shape *> vps; // What for?
4 Vector<Object *> vo;
5 Vector<Node *> vn;
```

Listing 3.7 – Différentes instanciations d'un modèle de classe Vector.

Dans le cas des déclarations du listing 3.7 on peut se poser par exemple une question au sujet de la destruction d'un vecteur d'entier, par rapport à celle d'un vecteur de pointeurs sur des formes. En effet, le vecteur dans ce dernier cas doit-il s'occuper de la désallocation individuelle des formes pointées ? Comme mentionné dans le paragraphe précédent, nous serions alors dans la situation ou l'opérateur `delete` n'a effectivement pas de sens pour un entier, alors qu'on souhaite l'utiliser dans le cas d'un vecteur de pointeurs. Il est alors utile de pouvoir définir deux modèles, l'un pour les types *pointeur sur quelque chose* et un autre pour tous les autres types. Cet exemple est très courant. Qui plus est, une solution élégante s'applique à ce cas précis : Elle consiste à utiliser l'héritage afin de minimiser le code supplémentaire à écrire. En effet, seul le destructeur est particulier dans le cas des pointeurs alors que tout le reste peut être défini pour le type `void*` comme pour un `int`. On réalise donc :

1. Une spécialisation *complète* du modèle pour le type `void*`, puis
2. une spécialisation *partielle* du modèle pour le type T*.

Toute ceci est illustré par le listing 3.8 qui montre au passage la syntaxe de la spécialisation, qui peut être schématisée de la manière suivante :

```
1  class Complex;
2
3  // (1) Forward declaration of Matrix
4  template <typename T> class Matrix;
5
6  // (2) Forward declaration of the operator template
7  template <typename T>
8  Matrix<T> operator+(const Matrix<T> &, const Matrix<T> &);
9
10 template <typename T>
11 class Matrix {
12   T ** data;
13   unsigned int rows, columns;
14   T ** allocate(unsigned long, unsigned long){};
15 public:
16   Matrix() {
17     data = 0;
18   }
19   Matrix(const Matrix &);
20   Matrix(unsigned long l, unsigned long c)
21     : rows(l), columns(c) {
22     data = allocate(l, c);
23   }
24   ~Matrix();
25   Matrix & operator=(const Matrix &);
26   Matrix operator*(const Matrix &);
27   // (3) friend function template
28   friend Matrix<T> operator+<>(const Matrix<T> &,
29                                const Matrix<T> &);
30   T * operator[](unsigned long);
31 };
32
33 template <typename T>      // (4) template definition
34 Matrix<T> operator+(const Matrix<T> & a,
35                     const Matrix<T> & b) {
36   // ...
37 }
38
39 int main() {
40   Matrix<int> m1(10, 2);
41   Matrix<float> m2(12, 12);
42   Matrix<Complex> m3(12, 23);
43 } // ...
```

Listing 3.6 – Exemple de modèle de classe avec une fonction amie.

Modèle général

```
template<typename T, typename U,...> class NomModèle {
  //...
};
```

Spécialisation complète

```
template<> class NomModèle<idTypePourT,idTypePourU,...> {
  //...
};
```

Spécialisation partielle

```
template<typename U,...> class NomModèle<idTypeSpecialisePourT,U,...> {
  //...
};
```

(Viennent après le mot-clé *template* tous les noms de paramètres de types qui font que la spécialisation n'est pas complète. Ils sont aussi répétés à droite du nom du modèle.)

Définition d'une méthode dans le cas d'une spécialisation complète

```
typeRetour NomModèle<Type1,Type2,...>::idMéthode(Type1 arg1,...) {
  // Code de la méthode
}
```

Notez l'absence du mot-clé `template<>` au début du prototype. Il s'agit alors en effet d'une simple définition.

3.4 Les modèles de fonctions

De même qu'il est possible de définir des modèles de classes (donc des modèles de méthodes), on peut définir des modèles de fonctions. Comme pour les classes, un tel modèle permet de générer automatiquement le code d'une fonction adaptée à un ou des types particuliers à partir d'un *patron* de fonction.

Un modèle est instancié (c.-à-d., du code est généré) lorsque le compilateur rencontre un appel de fonction modèle avec des arguments de types définis. La syntaxe de cet appel est en tout point similaire à un appel de fonction classique, sauf lorsque le type des arguments du modèle ne peut être déduit. C'est le cas des fonctions sans arguments, mais aussi lorsqu'aucun paramètre de type n'est utilisé dans la liste d'arguments, ou encore lorsqu'il y a ambiguïté en raison de l'application possible de conversions implicites multiples.

3.4.1 Syntaxe

Un modèle de fonction est défini de la manière suivante :

```
1  template <typename T>
2  class Vector {
3    // ...
4  };
5
6  template <>
7  class Vector<void *> {
8    // ...
9    void * & operator[](int i);
10 };
11
12 Vector<void *> v;
13
14 template <typename T>
15 class Vector<T *> : private Vector<void *> {
16   typedef Vector<void *> Base;
17   // ...
18   T * & operator[](int i) {
19     return reinterpret_cast<T * &>(Base::operator[](i));
20   }
21 };
```

Listing 3.8 – Spécialisations complète et partielle d'un modèle de classe.

```
template<typename T0, typenameT1,[...]> T0 idFonction ( T1 arg1, [...]) {
   [...]
};
template<typename T0,[...]> type idFonction( T0 arg1, [...]) {
   [...]
};
template<typename T0, type1 arg,[...]> type idFonction ( T0 arg1, [...]) {
   [...]
};
template<typename T0, type1 arg,[...]> type idFonction ( [...]) {
   Utilisation des types ou valeurs paramètres.
   [...]
};
```

L'appel peut être totalement transparent (c.-à-d., aucune différence par rapport à une fonction classique), il peut aussi rendre explicite le type ou les valeurs des paramètres du modèle :

```
idFonction(arg1, [...]);
idFonction <type1,type2,[...]>();
idFonction<type1,type2, constante1,[...]>(arg1, [...]);
```

La surcharge de fonctions autorise enfin une forme de spécialisation...

3.4.2 Surcharge

Contrairement aux modèles de classes, la surcharge des modèles de fonctions par une fonction classique est possible. Au moment de l'appel, si les types ne sont pas précisés (entre <>) et qu'il y a correspondance stricte des types, la priorité est donnée aux fonctions ordinaires. Ensuite, les spécialisations les plus précises sont recherchées. (Un paramètre de type déduit ne peut faire ensuite l'objet d'une promotion, d'une conversion standard, ou d'une conversion définie par l'utilisateur.)

3.5 Remarques finales

— Une classe peut posséder des modèles de méthodes ;
— Un modèle de classe peut aussi posséder des modèles de méthodes (c.-à-d., qui ont leurs propres paramètres de types).