



Centre **I**nformatique pour les **L**ettres
et les **S**ciences **H**umaines

Apprendre C++ avec Qt : Leçon 18 Outils de programmation générique

1 - Patrons de fonctions.....	2
Définition de patrons de fonctions.....	2
Utilisation de fonctions définies par un patron.....	3
Spécification implicite des paramètres de patron.....	4
Patrons recevant des valeurs comme paramètres.....	4
Spécialisation explicite.....	5
Fonctions membre définies par un patron.....	6
2 - Patrons de classes.....	6
Définition de patrons de classes.....	6
Instanciation de classes définies par un patron.....	6
Valeurs par défaut des paramètres de patrons de classes.....	7
Fonctions membre de classes définies par un patron.....	7
3 - Où doit-on définir les patrons ?.....	8
5 - Bon, c'est gentil tout ça, mais ça fait quand même 8 pages. Qu'est-ce que je dois vraiment en retenir ?.....	8

Une des caractéristiques majeures du langage C++ est l'importance qu'il accorde à la création de nouveaux types de données, spécifiquement adaptés au problème traité par un programme particulier. Cette approche soulève une difficulté lorsqu'on cherche à écrire du code très général, que l'on souhaite pouvoir réutiliser (sans avoir à y apporter la moindre modification) dans des projets très divers. En effet, les types utilisés par ces projets ne sont pas seulement différents et nombreux, ils sont indéterminés au moment où l'on écrit le code général. L'écriture de code prenant en charge cette quasi-infinité de types est ce qu'on appelle la programmation générique, qui est rendue possible en C++ par les patrons (ou, en anglais, *templates*).

1 - Patrons de fonctions

Un patron de fonction est une sorte de "fonction potentielle" : c'est un canevas, un plan à partir duquel le compilateur est capable, en fonction des besoins, de générer plusieurs fonctions réelles, qui diffèrent par le type des objets qu'elles manipulent.

Définition de patrons de fonctions

Puisque le patron doit pouvoir être finalement "concrétisé" avec différents types, sa définition doit utiliser une "abstraction" du type, ou, en d'autres termes, doit permettre de convenir d'un nom permettant de désigner le type sans avoir à dire de quel type il s'agit.

C'est un peu le même problème que dans le cas d'une fonction qui doit être capable de travailler sur plusieurs valeurs : l'utilisation d'un paramètre procure une abstraction de la valeur, ce qui permet à un programmeur ignorant tout de cette valeur de la désigner en utilisant simplement le nom du paramètre. Les patrons peuvent être décrits comme un moyen de paramétrer les types, et non simplement les valeurs.

La syntaxe permettant d'introduire cette "abstraction de type" est assez simple : il faut annoncer que **la définition qui va suivre est celle d'un patron**, puis indiquer entre chevrons les **"arguments de patron"** qui vont être utilisés. La déclaration d'un argument de patron ressemble beaucoup à celle d'un argument de fonction : le nom choisi est simplement précédé d'une indication sur la **nature de l'objet** désigné. Comme l'objet en question n'est ni une valeur, ni une variable, ni quoi que ce soit de ce genre, on utilise un mot spécial, `typename`, qui indique clairement que le nom en cours de définition désignera un type.

```
/ template <typename unType>
```

Le mot `typename` peut être remplacé par le mot `class`, mais cette tolérance (conservée pour des raisons purement historiques) nuit à l'intelligibilité du code, puisque le type en question peut très bien ne pas être une classe mais un type prédéfini ou énuméré.

Une fois posé ce préambule, la définition d'un patron de fonction ne se distingue de la définition d'une véritable fonction que par le fait qu'elle peut utiliser le nom de l'argument de patron comme s'il s'agissait d'un type existant réellement. Ceci signifie que les paramètres, les variables locales et la valeur renvoyée par la "fonction potentielle" peuvent relever d'un type inconnu du programmeur définissant le patron.

```
2 unType * metaFonction(unType & parametre) //reçoit une référence
3 {
4     unType variableLocale;
5     return & parametre; //renvoie adresse de...
6 }
```

Cet exemple, dépourvu de toute signification, ne prétend qu'illustrer diverses possibilités :

- un paramètre de type "référence à un objet d'unType qui sera précisé le moment venu" ;
- une variable locale "d'unType qui sera précisé le moment venu" ;
- le renvoi d'une valeur du type "adresse d'un objet d'unType qui sera précisé le moment venu"

Tout comme les fonctions, les patrons peuvent disposer de plusieurs arguments. Lorsque c'est le cas, la "fonction potentielle" dispose de plusieurs "types qui seront précisés le moment venu", ce qui élargit d'autant ses possibilités quant aux types de ses paramètres, de ses variables locales et de la valeur qu'elle renvoie. L'exemple suivant est un patron à deux paramètres définissant potentiellement des fonctions qui sont, pour leur part, dépourvues de paramètres :

```

1 template <typename unType, typename unAutre>
2     bool typesDeLaMemeTaille()
3     {
4         return sizeof(unType) == sizeof(unAutre);
5     }

```

Utilisation de fonctions définies par un patron

Il n'est bien entendu pas question d'utiliser réellement une fonction qui n'est que "potentielle". L'appel d'une fonction définie par un patron exige donc que le compilateur génère effectivement une fonction à partir du patron en question, et c'est à ce moment là qu'il faut préciser le(s) type(s) que l'on s'est abstenu de mentionner concrètement dans la définition.

La syntaxe employée pour spécifier les **paramètres de patrons** est étroitement inspirée de celle utilisée pour les paramètres de fonctions : le **nom du patron** est suivi d'un couple de crochets à l'intérieur desquels prendra place une **liste de types**. Ceci fait, il reste encore à mentionner, entre parenthèses, les valeurs destinées à initialiser les **paramètres de la fonction** qui va être générée à partir du patron. Le patron défini dans l'exemple précédent pourrait donc être utilisé ainsi :

```

1 bool b1 = typesDeLaMemeTaille < int, char *> ();
2 bool b2 = typesDeLaMemeTaille < bool, double> ();

```

Le premier appel exige que le compilateur **instancie le patron** pour générer et exécuter la fonction

```

1 bool typesDeLaMemeTaille()
2 {
3     return sizeof(int) == sizeof(char *);
4 }

```

alors que le second exige la création et l'appel de

```

1 bool typesDeLaMemeTaille()
2 {
3     return sizeof(bool) == sizeof(double);
4 }

```

Comme dans le cas des constructeurs et des destructeurs générés par défaut, la création par le compilateur de fonctions basées sur un patron ne se traduit pas par l'écriture automatique de code source qui serait ensuite compilé, mais par la création directe de code exécutable. Les deux fonctions évoquées ci-dessus n'existent donc jamais sous la forme visible présentée ci-dessus, ce qui est heureux, car elles seraient alors mutuellement exclusives, puisque homonymes et pourvues de signatures identiques...

Si l'on fait abstraction de sa totale inanité, notre premier exemple pourrait être utilisé ainsi :

```

1 int unEntier;
2 int * ptr = metaFonction <int> (unEntier);

```

ce qui se traduirait par la création et l'appel de

```

1 int * metaFonction(int & parametre)
2 {
3     int variableLocale;
4     return & parametre;
5 }

```

Les paramètres de patrons de fonctions ne peuvent pas être dotés de valeurs par défaut.

En d'autres termes, il n'est pas possible d'écrire quelque chose comme :

```

1 template <typename unType, typename unAutre = int> //ERREUR !
2     bool typesDeLaMemeTaille()
3     {
4         return sizeof(unType) == sizeof(unAutre);
5     }

```

Il faut donc nécessairement que les types paramétrés soient spécifiés lors de l'instanciation du patron, soit explicitement (comme dans les exemples précédents), soit implicitement.

Spécification implicite des paramètres de patron

Le patron `metaFonction <>` présente une particularité que l'on retrouve fréquemment dans des patrons plus utiles : le paramètre du patron sert à indiquer le type du paramètre des fonctions potentielles. Lors de l'utilisation d'un tel patron, l'information est donc redondante, puisque le type mentionné entre les crochets est celui de la valeur mentionnée entre les parenthèses.

Dans cette situation, le langage C++ accepte de se charger lui-même de la spécification du (ou des) paramètre(s) du patron, et les crochets, devenus inutiles, ne figurent plus dans l'appel. Le patron `metaFonction <>` pourrait donc être utilisé ainsi :

```
1 int unEntier;
2 int * ptr = metaFonction(unEntier); //équivalent à
//int *ptr = metaFonction <int>(unEntier);
```

Lorsque le patron comporte plusieurs paramètres, il est possible qu'ils soient tous spécifiés implicitement par les types des arguments passés à la fonction, mais ce n'est pas nécessairement le cas. Toutefois, lorsqu'on utilise une spécification implicite partielle, il est nécessaire que celle-ci porte sur les derniers paramètres du modèle. En d'autres termes, dès lors qu'un paramètre est omis, tous les paramètres suivants doivent l'être aussi.

Nous retrouvons ici la logique qui prévaut pour l'utilisation des valeurs par défaut des paramètres de fonctions. Etant donné que la correspondance entre les paramètres et les informations fournies lors de l'appel ne repose que sur leur ordre, il n'est pas possible de reprendre la spécification explicite après avoir accepté des informations implicites.

La spécification implicite d'un paramètre de patron n'est, bien entendu, possible que si au moins un des paramètres de la fonction générée est du type en question. Si, par exemple, un patron comporte **un paramètre fixant le type de la valeur renvoyée** et non celui d'un paramètre, il ne pourra être utilisé qu'au prix d'une spécification au moins partiellement explicite. Un patron défini ainsi :

```
1 template <typename typeRetourne, typename typeParametre>
2     typeRetourne laFonction(typeParametre leParametre)
3     {
4         typeRetourne resultat;
5         //...
6         return resultat;
7     }
```

pourra donc être utilisé soit en spécifiant **explicitement** ses deux arguments, soit en spécifiant **implicitement** son second argument :

```
1 int k = laFonction <int, double> (2); //int laFonction(double leParametre)
2 bool x = laFonction <bool> (2);      //bool laFonction(int leParametre)
```

L'omission des chevrons élimine toute différence syntaxique entre l'appel d'une fonction ordinaire et l'appel d'une fonction générée à partir d'un modèle. Par conséquent,

Un programmeur qui utilise des fonctions générées à partir de patrons qui lui ont été fournis et qui permettent de spécifier implicitement tous leurs arguments peut assez facilement oublier que les fonctions en question sont générées à partir d'un patron, voir même ignorer totalement la notion de patron.

Patrons recevant des valeurs comme paramètres

Les patrons que nous avons décrits jusqu'ici n'utilisent que des types comme paramètres. Si cette possibilité constitue la caractéristique majeure des patrons, il arrive cependant que certains patrons aient recours à des paramètres qui permettent de spécifier une valeur décrivant un aspect des fonctions qui seront générées à partir du patron. L'exemple suivant montre un patron

permettant de générer des fonctions dont une constante locale est initialisée avec différentes valeurs, et qui comportent des (vrais) tableaux locaux de tailles différentes :

```
1 template <typename T, int taille>
2 void creeTableau()
3 {
4     const int c = taille; //une constante locale paramétrée
5     T tableau[taille];    //un vrai tableau local
6 }
```

Lors de l'instanciation d'un tel patron, la valeur transmise doit nécessairement être une **constante** (c'est à dire une valeur connue lors de la compilation). En effet, le code de la fonction répondant à l'appel qui cause cette instanciation doit être généré lors de la compilation, et tous les paramètres de patron doivent donc être spécifiés dès cet instant. On pourra donc écrire :

```
creeTableau <double, 1024> ();
```

mais pas

```
1 int nb = 1024;
2 creeTableau <double, nb> (); //ERREUR : il faut une constante !
```

NB. Les paramètres de patrons de fonctions qui sont des valeurs plutôt que des types sont, eux aussi, privés de la possibilité de disposer de valeurs par défaut.

Spécialisation explicite

Il arrive que certains types exigent un traitement dérogatoire par rapport au cas général. Il reste alors possible d'utiliser un patron de fonction, car le langage autorise la définition explicite de la version "exceptionnelle" de la fonction. Un exemple simple et classique est la détermination de la plus petite de deux valeurs du même type. Il s'agit d'un cas typique où un modèle de fonction s'impose, puisqu'il permet de traiter un grand nombre de cas avec une seule ligne de code :

```
1 template <typename unType>
2 unType mini(unType v1, unType v2)
3 { return (v1 < v2) ? v1 : v2; }
```

A partir de ce patron, le compilateur est en mesure de générer les fonctions appropriées pour tous les types numériques prédéfinis, ainsi que pour les classes disposant de l'opérateur de comparaison nécessaire (<, en l'occurrence). Il reste toutefois un cas tout à fait digne d'intérêt qui n'est pas pris en charge correctement : si l'utilisateur cherche à comparer deux tableaux de caractères, il souhaite vraisemblablement que la fonction `mini()` lui indique laquelle des deux chaînes précède l'autre dans l'ordre alphabétique. Or, à la suite de

```
1 char * chaines[] = {"Zèbre", "Antilope"};
2 char * premier = mini(chaines[0], chaines[1]);
```

la fonction qui est générée et exécutée est

```
1 char * mini(char * v1, char *v2)
2 { return (v1 < v2) ? v1 : v2; }
```

L'adresse renvoyée est donc la plus petite des deux, ce qui n'a rien à voir avec l'ordre alphabétique des chaînes stockées à ces adresses. On corrige ce problème en "spécialisant" le patron, c'est à dire en fournissant la définition de la fonction qui doit être exécutée lors d'un appel qui exigerait l'instanciation du patron avec le type qui pose problème. La syntaxe de cette "**spécialisation explicite**" ressemble à la définition d'un patron de fonction dépourvu de paramètre : les chevrons restent vides. Le lien entre le patron et sa spécialisation explicite est simplement assuré par le fait que les fonctions décrites portent le même nom et ont des signatures "isomorphes".

```
1 template <>
2 char * mini(char * v1, char * v2)
3 {
4     int i = 0;
```

```

5   while (v1[i] == v2[i] && v1[i] != 0)
6       ++i;
7   return (v1[i] < v2[i]) ? v1 : v2;
8   }

```

Une fois cette spécialisation définie, un appel de la fonction `mini()` utilisant des chaînes de caractères pour spécifier la valeur des paramètres ne se traduira pas par une instantiation du patron, mais par l'exécution de la version "spécialisée" que nous avons écrite nous-mêmes.

Fonctions membre définies par un patron

Un patron peut également servir à générer des fonctions membre plutôt que des fonctions globales. Les seules restrictions qu'il faut alors accepter sont que les paramètres de ces fonctions doivent être dépourvus de valeurs par défaut, et que **ces fonctions ne peuvent pas être virtuelles**. La mise en œuvre d'un "patron membre" ne présente aucune particularité syntaxique :

```

1  class CExemple
2  {
3  public:
4      template <typename unType>
5          int test(unType p) {return sizeof(unType);}
6  };

```

Les fonctions membres nécessaires sont alors automatiquement générées pour satisfaire l'usage qui est fait de la classe. Une séquence telle que

```

1  CExemple unExemple;
2  unExemple.test(4);

```

se traduira donc par la génération de la fonction

```
int CExemple::test(int p) {return sizeof(int); }
```

2 - Patrons de classes

On peut donner des patrons de classes une définition tout à fait analogue à celle que nous avons donné des patrons de fonctions : un patron de classe est une sorte de "classe potentielle", c'est à dire un canevas, un plan à partir duquel le compilateur est capable, en fonction des besoins, de générer plusieurs classes réelles, qui diffèrent par le type des objets qu'elles impliquent.

Définition de patrons de classes

La syntaxe de la définition d'un patron de classes est semblable à celle de la définition d'un patron de fonction : un **préambule** indique qu'il s'agit d'un patron et introduit les **identificateurs des paramètres** de patrons, qui peuvent être soit des **types** soit des **valeurs**. Ces identificateurs peuvent ensuite être utilisés dans la définition du patron de classe, définition qui ressemble par ailleurs tout à fait à celle d'une véritable classe :

```

1  template <typename unType, int n >
2      class CMetaClasse
3      {
4      public:
5          unType membre;
6          char tableau[n];
7      };

```

Instantiation de classes définies par un patron

Bien entendu, l'instanciation d'une classe définie par un patron exige la spécification des paramètres devant être utilisés par celui-ci. Le patron de classes défini ci-dessus pourrait, par exemple, être utilisé ainsi :

```
CMetaClasse < int, 12 > uneInstance;
```

La présence d'une telle définition dans un programme force le compilateur à définir une classe comportant un membre de type `int` et un tableau composé de douze `char`. Cette classe est le type adopté pour la variable nommée `uneInstance`.

Cette définition provoque donc une double instanciation : le patron de classe est instancié pour définir une classe, qui est ensuite elle-même instanciée pour donner naissance à la variable.

Valeurs par défaut des paramètres de patrons de classes

A la différence des patrons de fonctions, les patrons de classes peuvent adopter des valeurs par défaut pour leurs paramètres. On pourrait par exemple écrire

```
1 template <typename unType = bool , int n = 4>
2   class CMetaClasse
3   {
4   public:
5       unType membre;
6       char tableau[n];
7   };
```

ce qui permettrait d'instancier la classe sans avoir à préciser les paramètres :

```
CMetaClasse < > maVariable; //équivalent à CMetaClasse <bool, 4 > maVariable;
```

Bien que l'existence des valeurs par défaut nous dispense de fournir des paramètres, le couple de crochets reste indispensable pour identifier `CMetaClasse` comme étant le nom d'un patron.

Fonctions membre de classes définies par un patron

Toutes les fonctions membre d'une classe définie par un patron sont nécessairement elles-mêmes définies par un patron.

Cette caractéristique étant inéluctable, la syntaxe de la définition de la classe tolère qu'elle ne soit pas spécifiée explicitement. On écrira donc :

```
1 template <typename unType>
2   class CMetaClasse
3   {
4   public:
5       void uneFonctionMembre();
6   };
```

Si on y tient absolument, on peut [rappeler le statut "patronesque"](#) de ces fonctions membre :

```
template <typename unType>
class CMetaClasse
{
public:
    template <typename unType> //oiseux, mais OK
        void uneFonctionMembre();
};
```

Si la définition de la fonction membre est rejetée à l'extérieur de la définition de la classe, en revanche, aucun implicite n'est possible et le fait que **cette fonction est définie par un patron** doit être annoncé clairement :

```
1 template <typename unType>
2   void CMetaClasse<unType>::uneFonctionMembre()
3   {
4       //...
```

Comme à l'accoutumée, la définition d'une fonction membre à l'extérieur de la définition de la classe exige qu'on mentionne le nom complet de la fonction (cf. [Leçon 3](#)), ce qui nécessite de **nommer la classe**. Etant donné qu'il s'agit ici d'une classe définie par un patron, il convient de

préciser quelle valeur du **paramètre de patron** de classe doit être utilisée pour créer la classe dont cette fonction est membre. Cette valeur doit bien entendu correspondre à **celle pour laquelle on est en train de créer la fonction...**

Cette exigence vaut également pour les éventuels constructeurs et destructeurs définis hors de la classe, ce qui, dans notre exemple, se traduirait par :

```
1 template <typename unType>
2   CMetaClasse<unType>::CMetaClasse() //constructeur par défaut
3   {
4     //...
5   }
```

```
1 template <typename unType>
2   CMetaClasse<unType>::~~CMetaClasse() //destructeur
3   {
4     //...
5   }
```

3 - Où doit-on définir les patrons ?

La définition d'un patron se rapproche de celle d'un type en ceci qu'elle ne donne pas lieu directement à la génération de code (ou à la réservation de mémoire) lors de la compilation. C'est seulement lorsque le patron ou le type sont instanciés qu'il y a réservation de mémoire et/ou génération de code. Cette remarque suggère que, comme les types,

Les patrons doivent être définis dans des fichiers .h

Il suffit alors de veiller à inclure ce fichier .h en tête des fichiers source contenant des instanciations du patron en question. Cette approche a le mérite de la simplicité, mais elle présente deux inconvénients lorsque le code concerné devient complexe et volumineux :

- le code source étant placé dans le .h, il est accessible à tous les utilisateurs du patron ;
- la durée de compilation du projet peut parfois se trouver dramatiquement accrue.

Lorsque ces inconvénients deviennent réellement pénalisants, il est possible d'adopter une approche différente, et de séparer la déclaration des patrons (dans un .h) de leur définition (dans un .cpp). Ces définitions doivent alors être précédées du mot **export**. Les complications introduites par cette stratégie ne sont rentables que dans le cas de projets relativement avancés.

5 - Bon, c'est gentil tout ça, mais ça fait quand même 7 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) Un patron de fonction est un "moule" qui permet au compilateur de fabriquer, en fonction des besoins, toute une collection de fonctions appliquant un même traitement à des données de types différents.
- 2) Pour appeler une fonction définie par un patron, il faut spécifier les types qu'elle est supposée traiter.
- 3) Une fonction définie par un patron ne peut pas être virtuelle.
- 4) Un patron de classe est un "moule" qui permet au compilateur de fabriquer, en fonction des besoins, toute une collection de classes appliquant un même ensemble de traitement à des données de types différents.
- 5) Pour instancier une classe définie par un patron, il faut spécifier les types qu'elle est supposée traiter.
- 6) Toutes les fonctions membre d'une classe définie par un patron sont elles-même définies par des patrons de fonctions.
- 7) Les patrons de fonctions ou de classes doivent être définis dans des fichiers .h
- 8) Une fois un patron défini, il est facilement utilisable, y compris par des programmeurs ignorant tout des subtilités de la programmation générique.

Au fait, combien avez-vous écrit de programmes utilisant des `QMap` <> et des `QValueList` <> avant d'étudier la Leçon 18 ?