# *partie 6 : Patrons, flots, chaînes de caractères et exceptions*

**MOOC Langage C++ INSA**



Source : Ce document est en partie extrait du livre : du langage C au C++ par Thierry Monteil, Vincent Nicomette, François Pompignac, Saturnino Hernando, Presses Universitaires du Midi ISBN 978-2-8107-0054-7



## 1 Les patrons

Les patrons ou "template" permettent de paramétrer les classes et les fonctions par des types. Ceci met en œuvre le concept de généricité. Ce dernier se retrouve aussi dans des langages comme ADA. Ceci s'utilise par exemple pour :

- les types paramétrés : on peut alors définir par exemple des listes chaînées pour n'importe quel type. Ceci est intéressant car les opérations sur les listes sont indépendantes des types stockés,
- une fonction : par exemple pour un algorithme de tri de tableau, l'algorithme mis en œuvre est toujours le même pour n'importe quel type pourvu que l'on possède une fonction de comparaison pour un type donné.

C'est le mot clé `template` qui va permettre d'exprimer en C++ la généricité.

### 1.1 Fonction paramétrée par un type

La définition d'une fonction paramétrée par un type se fait simplement en indiquant le paramètre générique de la fonction. Voici un exemple, où l'on suppose que la fonction `quicksort` permet de réaliser un tri d'un tableau par l'algorithme quicksort.

**Exemple:**

```
template <typename T> void quicksort(T* tab, size_t n){
     ...
     if(tab[i]<tab[j]) ... // l'opérateur < est défini sur T
     ...
};
class element {
     ...
     public:
          int operator<(element&autre);// la définition de l'opérateur <
          /* est nécessaire si ce type ne posséde pas déjà cet opérateur*/
     ...
};
void essai (){
     int tab_int[100];
```

```
    element tab_ord[20];
    quicksort(tab_int,100);
    quicksort(tab_ord,20);
}
```

C'est le type des paramètres qui va instancier la bonne version de la fonction en tenant compte de la bonne version de l'opérateur de comparaison.

## 1.2 Classe paramétrée

La classe peut être paramétrée par un type mais aussi par d'autres arguments : **Exemple:**

```
template <typename T, int sz, void(*err)()> class table {
    T tab[sz];
    public:
        T& operator [] (int i){
            if (i ≥ sz || i<0) err();
            return tab[i];
        }
    ...
};
```

Dans cet exemple, le premier paramètre est un type, le deuxième un entier et le troisième un pointeur sur une fonction.

On peut définir des objets à partir de la classe : **Exemple:**

```
void f() {
    cerr << ''erreur d'indice'';      // traitement de l'erreur
}
int main(){
    typedef table<int,5,f> T5;
    T5 t0;
    table<T5,10,f> tab;
    table<int,10,f> tt[10],*pt;
    t0[0]=1;
    tab[1]=t0;
    ...
}
```

La classe générique se manipule comme une classe normale dès que ses paramètres génériques sont définis. Ces derniers doivent être de type constant car leur évaluation se fait à la compilation.

**Remarques:**

— le patron n'est pas un type. On ne peut pas déclarer une variable du type du patron sans avoir précisé les paramètres :

```
table t1; // ne compile pas
```

— `typename` peut être remplacé par `class`. Dans tous les cas, c'est un type ou un nom de classe qui est attendu,

— les patrons font appel à des mécanismes particuliers. La compilation séparée est impossible pour eux, tel que déclaré précédemment,

— Il est possible d'hériter d'une classe paramétrée :

**Exemple:**

```
...
class table2: table<class T, int sz, void(*err)()> {
     // T est utilisé comme un type classique
};
int main(){
     table2<int,5,f> t3;
     ...
}
```

**Remarque:**
Quelques mécanismes annexes permettent aux patrons de s'adapter à des cas particuliers. On peut par exemple faire de la spécialisation en précisant différents codes dans les méthodes en fonction de la valeur des paramètres du patron lors de son instanciation. On peut aussi donner des valeurs par défaut aux paramètres du patron.

# 2 Les flots

## 2.1 Introduction

La gestion des flots est une application de ce que peut permettre la programmation objet. Les flots ou "stream" (en anglais) permettent de gérer les entrées-sorties. L'avantage de cette notion est d'intégrer dans un schéma identique les manipulations des types standards et des types utilisateurs. Ce chapitre va décrire succinctement la bibliothèque de gestion des flots. Pour pouvoir l'utiliser, il faut inclure le fichier "iostream".

## 2.2 Sorties

C'est la classe `ostream` qui gère les sorties pour les types prédéfinis. Elle définit entre autres l'opérateur << pour ces types. L'utilisateur peut enrichir cette classe pour permettre la gestion des sorties pour ses classes.

**Exemple:**

```
ostream& operator << (ostream& os, const float_pair& x){
     return os << '(' << x.first() << ',' << x.second() << ')';
}
ostream& operator << (ostream& os, const complexe& x){
     if (x.imag() == 0) {return os << x.reel();}
     else {
          if (x.reel() == 0){
               return os << x.imag() << "*i";}
          else {
               return os << '(' << x.reel() << '+' <<
               x.imag() << "*i" <<')';
          };
     };
}
```

Le passage de paramètre se fait par référence. Le premier paramètre est le support de sortie. Le retour de la fonction indique la manière dont les données doivent être écrites sur le flot de sortie.

## 2.3 Entrées

L'entrée est similaire à la sortie. La classe utilisée est `istream`. Elle fournit l'opérateur d'entrée ">>" pour les types de base. Comme pour les sorties, l'utilisateur peut gérer la saisie de ses classes de façon standard.

**Exemple:**

```
istream& operator >> (istream& s, float_pair& x)
{
     float re, im = 0;
// saisie partie réel puis imaginaire suivi d'un retour chariot
     s>>re>>im;
     x = float_pair(re, im);
     return s; }
```

Un ensemble d'opérations permettent de tester le caractère lu :

```
int isalpha(char) // 'a'..'z' 'A'..'Z'
int isupper(char) // 'A'..'Z'
int isdigit(char) // '0'..'9'
int isspace(char) /* ' ' '/t' 'retour chariot' 'passage à la
ligne' 'saut de page' */
...
```

## 2.4 Etat du flot

Chaque flot "istream" et "ostream" possède un état. Les erreurs et les conditions non standards sont gérées en positionnant et en testant cet état de façon adéquate. L'état est représenté par la classe `ios`. Les opérations suivantes sont permises :

```
int eof() const; // fin de fichier
int fail() const; // la prochaine opération échouera
int bad() const; // le flot est altéré
int good() const; // la prochaine opération peut réussir
int clear(int i=0); // remet l'état du flot à "good"
```

**Exemple:**

```
istream& operator >> (istream& s, float_pair& x)
     /* formats d'entrée pour un float_pair; 'f' indique un flottant
     f ou bien ( f ) ou bien ( f , f ) ou bien ( f , f , */ {
     float re, im = 0;
     char c = 0;
     s >> c;
     if (c == '(') {           s >> re >> c;
          if (c == ',') s >> im >> c;
          if ((c != ')') && (c != ',')) s.clear(ios::badbit);
          //positionne l'état, vu plus loin
     }
```

```
else {//c est un signe ou un digit. Le remettre dans s
     s.putback(c);
     s >> re;
}
if (s) x = float_pair(re, im);
return s; }
```

## 2.5 Format

Il existe deux types de manipulation :

— configuration par l'utilisation de méthodes,

— configuration par un système de bascule et de valeurs spéciales insérées dans le flot (manipulateurs).

**Exemples:**

- configuration de la précision :

```
float f=3.1415926;
cout << setprecision (5) << f << endl;
```

- format d'affichage : scientific, fixed :

```
float f=3.1415926;
cout << scientific << f << endl;
```

- nombre de colonne

```
float f=3.1415926;
cout.width(20);
cout  << f << endl;
```

## 2.6 Gestion de fichier

Les flots peuvent gérer des fichiers en entrée et en sortie. On utilise pour cela le fichier `fstream`.

**Exemple:**

```
#include <fstream>
#include <iostream>
#include <libc.h>
using namespace std;
void error(char*s, char* s2= ‘‘’’){
     cerr << s << ’ ’ << s2 << ’\n’;
     exit(0);
}
int main (int argc, char* argv[]){
     if (argc != 3) error(‘‘nombre d’argument incorrect’’);
     ifstream from(argv[1]); // fichier d’entrée
     if (!from) error(‘‘impossibilité d’ouvrir le fichier en entrée’’,
     argv[1]);
     ofstream to(argv[2]); // fichier de sortie
     if (!to) error(‘‘impossibilité d’ouvrir le fichier en sortie’’,
     argv[2]);
     char ch;
```

```
    while (from.get(ch)) to.put(ch);
    if (!from.eof() || to.bad()) error(''erreur étrange'');
    return 0;
}
```

Ce programme attend deux arguments qui sont le nom du fichier en entrée et le nom du fichier en sortie. Il réalise alors une copie du premier fichier dans le second. La lecture se fait caractère par caractère. Un test est réalisé à la fin pour vérifier que l'on est à la fin du fichier d'entrée et qu'il n'y a pas eu d'erreur sur le fichier de sortie. La boucle "while" s'arrêtera dès le changement d'état du flot d'entrée.

# 3 Chaîne de caractères

## 3.1 Introduction

L'objectif de la classe `string` est de fournir un moyen aisé de manipuler des chaînes de caractères. Elle fournit tout ce que l'on peut attendre : gestion transparente et dynamique, affectation, concaténation, recherche de sous-chaîne, insertion, suppression, etc.

Un objet string contient à un instant un ensemble quelconque de caractères. La notion de caractère de fin de chaîne n'existe pas comme en C. Cette classe est très proche de la classe "*vector < char >*" (voir chapitre **??**). On va disposer d'itérateur pour la manipuler. Pour pouvoir l'utiliser, il faut inclure `string`.

## 3.2 Constructeurs

La classe dispose de nombreux constructeurs :

```
string ch1; // chaîne vide
string ch2(10,'*'); /* chaîne de 10 caractères tous
égaux à * */
string ch3("coucou"); /* chaîne de 6 caractères initialisés
 avec coucou*/
char* mess="coucou";
string ch4(mess); /* chaîne de 6 caractères initialisés
avec coucou */
string ch5(ch4); /* chaîne initialisée avec une autre*/
...
```

## 3.3 Opérations

Voici des exemples d'opérations fournies sur la classe string :

- Les opérateurs << et >> sont définis et utilisent les mêmes conventions que le C++,
- la concaténation est réalisée avec l'opérateur +,
- recherche de la première occurrence d'une sous-chaîne dans une chaîne : ch3.find("ou"), renvoie de l'indice dans la chaîne,
- insertion dans une chaîne d'un caractère ou d'une sous-chaîne : ch3.inser(ch3.begin()+1, 'a'),
- suppression dans une chaîne d'un caractère ou d'une sous-chaîne : ch3.erase(ch3.begin(), 'o'),
- etc.

# 4 Les exceptions

Les exceptions sont une méthode pour réagir aux cas d'erreur dans un programme ou une bibliothèque. L'idée est qu'une fonction qui rencontre un problème qu'elle ne peut traiter `lance` une exception, en espérant que son appelant saura comment réagir. L'appelant a alors le choix de propager l'exception ou de réagir à cette dernière en l'interceptant. Le C++ reprend le même type de gestion qu'ADA.

## 4.1 Mise en oeuvre

La mise en œuvre se fait en trois temps :

- Tout d'abord, il y a l'endroit où l'on met en œuvre le contrôle d'erreurs par les exceptions. Ceci correspond à un bloc d'instructions commençant par l'instruction `try` qui indique que l'on essaie d'exécuter un segment de code. Les instructions de ce code vont faire appel à des fonctions,
- Une erreur peut se produire : ceci correspond à la deuxième étape. L'instruction qui va découvrir une erreur va `lancer` (en anglais "throw") l'exception. À ce moment-là, l'exécution du code est arrêté et l'on va sortir du bloc "try',
- Le troisième temps correspond à la capture (en anglais "catch") ou non de l'exception.

**Exemple:**

```
class vecteur {
     int* p;
     int sz;
     public:
          class range{}; // classe de l'exception
          int & operator[](int i];
          ... }
int & vecteur::operator[](int i){
     if (0≤i && i<sz) return p[i];
     throw range(); // cas d'erreur mauvais indice, exception levée
}
void do_something(){
     ...
     v[v.size+10];// erreur d'indice
}

   int main(){
     ...
     try { // bloc où les exceptions sont gérées
          do_something()
          ...
     }
     catch (vecteur::range){ // capture de l'exception
          // réaction à l'exception
     }
     ...
}
```

Dans cet exemple, l'exception est levée par l'accès à un mauvais indice dans la fonction `do_something`. C'est l'opérateur d'indice qui lève l'exception. Elle n'est pas traitée dans `do_something`, l'exception est donc propagée à l'appelant, c'est-à-dire le programme principal. Ce dernier traite bien le cas de l'exception `range`.

## 4.2 Nommage des exceptions

Il est possible de mettre plus d'informations dans l'exception. Dans l'exemple précédent, il peut être intéressant de connaître la valeur de l'indice qui a déclenché l'exception. Dans ce cas, il suffit de déclarer des attributs pour l'exception `range` et de nommer celle-ci au moment de sa capture pour accéder aux attributs mis en place par le constructeur : **Exemple:**

```
class vecteur {
public:
...
      class range{
            public:
                  int index;
                  range(int i): index(i) {}
      };

         int & vecteur::operator[](int i){
           if (0≤i && i<sz) return p[i];
           throw range(i); //cas d'erreur mauvais indice, exception levée
      }
...
};

   int main(){
      ...
      try { // bloc où les exceptions sont gérées
            do_something()
            ...
      }
      catch (vecteur::range r){ // capture de l'exception
            // réaction à l'exception
            cerr << ''mauvais index'' << r.index << '\n';
      }
      ...
}
```

Les exceptions peuvent aussi être déclarées comme faisant partie d'une énumération. **Exemple:**

```
enum matherr {overflow,underflow, zerodivide};
...
try {
      ...
}
catch (matherr m) {
      switch(m){
            case overflow:
                  ...
            case underflow:
                  ...
            case zerodivide:
                  ...
      }
}
```

## 4.3 Gestion standard des exceptions

## 4.4 Exceptions non interceptées

Si une exception est levée et pas interceptée, il y a appel de la fonction `terminate`. La fonction `terminate` exécute la dernière fonction passée en paramètre à `set_terminate()`. Par défaut, c'est un appel à la fonction `abort()`, ce qui a pour effet d'arrêter le programme.

## 4.5 Modification du comportement

On peut modifier le comportement suite à la levée de certaines exceptions. Par exemple, un `new` qui échoue lève l'exception `bad_alloc`. On peut associer son propre comportement suite à cette exception via la fonction set_new_handler(mafonction) où la signature de "mafonction" est : `void mafonction()`.

## 4.6 Restriction des exceptions à une fonction

Il est possible de préciser les exceptions qu'une fonction est susceptible de déclencher avec la notation suivante :

```
void fonction(...) throw(type1exception, type2exception, ...)
{...}
```

Si une autre exception est levée à l'exécution dans cette fonction, un appel à la fonction unexpected() est fait.

## 4.7 Exception standard

Le C++ définit la classe `exception`. Il est possible et même conseillé de définir ses propres exceptions à partir de cette classe. Ceci permet, en faisant un "catch" sur `exception`, de récupérer toutes les exceptions : celles du langage et celles définies par l'utilisateur. Il est alors conseillé de redéfinir par exemple la méthode :

```
const char* what() const {...}
```

Cette dernière renvoie une chaîne de caractères précisant la cause de l'exception.