Patrons

Les modèles / Patrons / Template

la traduction de template en français varie suivant les auteurs ...

Fonctions génériques

Une fonction générique ou modèle de fonction, est une fonction dans laquelle le type de certains arguments est paramétré.

La syntaxe la plus simple est

```
template <class paramètre> définition de fonction
```

Exemple:

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

On peut alors écrire

```
int i = max(34,45);
double d = max(4.5,3.4);
```

Le compilateur construit deux instances de la fonction max, de prototypes

```
int max(int, int)
double max(double, double)
```

Notes:

1. Le compilateur n'engendre le code d'une fonction avec des arguments de type donné **que lorsqu'il y a un appel** de cette fonction. Potentiellement, le nombre de fonctions définies par un patron est illimité.
2. Dans la définition d'un patron, le mot-clé `class` n'a pas sa signification habituelle: il indique qu'un paramètre de type va suivre.
3. Le compilateur ne fait pas les conversions standard. Ainsi, on ne peut pas écrire avec le patron ci-dessus:

```
char c;
int i,j;
...
j = max(i,c);
```

Il y a des patrons à plusieurs paramètres:

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

ce qui permet d'interpréter correctement le max(i,c) ci-dessus.

Classes génériques

Les classes génériques ou patrons ou modèles de classes permettent de définir des classes paramétrées par un type ou par une autre classe. Ceci permet d'obtenir une certaine généricité et est donc très utile pour les structures de données.

Par exemple, une pile d'entiers ne diffère d'une pile de caractères que par le type de ses éléments. Si l'on dispose d'une pile paramétrable, on peut écrire:

```
main()
{
  ...
  Pile <int> S;
  Pile <char> Op;
  ...
  char op = Op.pop();
  S.push(x*y);
  ...
}
```

Exemple: un programme rudimentaire d'évaluation d'expressions arithmétiques complètement parenthésées. Il utilise deux piles: une pile d'opérandes et une pile d'opérateurs.

```
void main() {
  char c;
  Pile <int> S;
  Pile <char> Op;
  while (cin.get(c)) {
    if (c == '\n') break;
    switch(c) {
      case '(': break;
      case '+':
      case '*':
        Op.push(c);
        break;
      case ')':
        int x = S.pop();
        int y = S.pop();
        switch (char op = Op.pop()) {
          case '+' : S.push(x+y); break;
          case '*' : S.push(x*y);
        };
        break;
      default :
        if (('0'<=c) && (c<='9'))
          S.push(c-'0');
    }
  }
  cout << S.pop();
}
```

Définition

Définition
Une classe générique ou modèle ou patron, en anglais template est une classe paramétrée par un type.

Exemple

```
template <class T> class Pile {
  private:
    T p[10];
    int sp;
  public
    Pile() {sp = 0;}
    void push(T x) { p[sp++] = x; }
    T pop(void) {return p[--sp]; }
};
```

On remarque
* Un nouveau mot-clé template;
* Le paramètre est entre chevrons;
* Définition de classe concrète par spécification d'un type.

```
Pile<int> S;
```

ou fréquemment

```
typedef Pile<int> PileEntiers;
PileEntiers S;
```

Plusieurs paramètres peuvent intervenir:

```
template <class T, int Max>
class Pile {
private:
  T p[1+Max];
  int sp;
  int maximum;
public
  Pile() : maximum(Max), sp(0) { }
  void push(T x) { p[sp++] = x; }
  T pop(void) { return p[--sp]; }
  int IsFull(void) { return sp == maximum;}
};
```

Si l'on sépare les déclarations des définitions, il convient de répéter le mot-clé template à chaque définition.

Ou des template template parameters

```
template <template <class> class ThreadingModel>
   struct RefCountedMTAdj
   {
       template <class P>
       class RefCountedMT : public ThreadingModel< RefCountedMT<P> >
       {
           typedef ThreadingModel< RefCountedMT<P> > base_type;
           typedef typename base_type::IntType       CountType;
           typedef volatile CountType                *CountPtrType;

       public:
           RefCountedMT() ;

           //MWCW lacks template friends, hence the following kludge
           template <typename P1> RefCountedMT(const RefCountedMT<P1>&
  rhs)
           : pCount_(reinterpret_cast<const RefCountedMT<P>&>(rhs).
  pCount_)
           {}

           bool Release(const P&);
           enum { destructiveCopy = false };

       private:
           // Data
           CountPtrType pCount_;
       };
   };
```

Concrétisation et spécialisation

```
template <class T, class U, int N>
class Patron {
  // ...
}
```

La définition d'un constructeur s'écrit, en dehors de la déclaration:

```
template <class T, class U, int N>
Patron<T, U, N>::Patron() {...}
```

On concrétise le patron en donnant des types ``concrets'':

```
class Patron<int, float, 5> C1;
class Patron<Point, int, 100> C2;
```

si `Point` est une classe. On peut substituer à un paramètre un autre patron.

On spécialise est redéfinissant une partie de la classe. Par exemple, pour un patron

```
template <class T>
class Point {
  // ...
}
```

on précise la définition dans le cas des caractères

```
class Point<char> {
  // redefinition ou extension
}
```

On peut ne définir qu'une seule méthode par

```
void Point<char>::doit(char)...
```

on parlera + tard de spécialisation partielles

Une pile générique

Voici une pile générique, implémentée à l'aide d'une liste chaînée.

```
template <class Arg>
class Pile {
public :
      Pile() { p = NULL; }
      void push(Arg x) { p = new Cellule(x, p); }
      Arg top() { return p -> cont; }
      Arg pop();
private :
  struct Cellule {
    Cellule* suiv;
    Arg cont;
    Cellule(Arg c, Cellule* s) : cont(c), suiv(s) {}
  };
  Cellule * p;
};
```

Pour une méthode définie hors de la déclaration (comme la méthode pop()), on doit répéter l'en-tête complet:

```
template <class Arg>
Arg Pile<Arg>::pop() {
  Cellule* q = p;
  Arg c = q -> cont;
  p = p -> suiv;
  delete q;
  return c;
}
```

Cette méthode réalise la suppression explicite de la cellule, mais bien entendu pas de l'argument.

Cast

Une traduction de Cast est Plâtre ! Qui permet d'immobiliser un membre :)

Le cast du C

```
(type) expression
```

Le même en C++

```
static_cast<type>(expression)
```

Les autres cast du C++ on une portée plus restreinte :

const_cast<type> retire la partie const du type : `const_cast<const int>` est un `int`
mais si

```
Truc a,c;
const Truc & b = a;
const_cast<Truc>(b)=c;
```

fonctionne par contre

```
const_cast<Truc *>(c)
```

ne fonctionne pas !
Le compilateur vérifiant que le seul effet de const_cast est de retirer le qualifier const.

Même chose pour volatile_cast qui ne doit pas servir souvent :) !

dynamic_cast reinterpret_cast

Permet de faire des conversions "sures" en montant ou en traversant des arbres d'héritage.
Les dynamic_cast s'utilise sur des pointeurs ou des références vers des classes de base pour les convertire en pointeurs ou références sur des classes dérivées ou voisines (rappel les cast dans l'autre sens sont implicites).
De plus le dynamic_cast vous permet de savoir si la conversion a échoué ou non.
Sur les pointeurs dynamic_cast est évaluer a NULL si il y a échec.
Sur les références il y a levé d'une exception `std::bad_cast` en cas d'échec .

Utile pour une fonction générique qui veut faire un traitement spécial pour un élément d'une classe dérivée.

Le reinterpret_cast est utiliser pour faire des conversions implémentations dépendantes par exemple un cast de pointeur de fonction renvoyant un int et sans paramètre en pointeur de fonction renvoyant void et sans paramètre :

```
typedef void (*Funcptr)();
int elvis();
Funcptr p=reinterpret_cast<Funcptr>(&elvis);
```

Le C++ n'offre pas de garanties sur la validité de ce cast (dommage cela permettrait de faire des multi-methodes in C++).

Implicite c'est le MAL

Implicite copy constructor
implicite operator =
implicite cast X(int){}

Cacher moi toutes les fonctions qui peuvent être appelées implicitement !
Soit en les rendant `private` soit pour les constructeurs a 1 argument en utilisant le mot clef explicite.

```
Class toto {
toto &operator =(const toto & u ){ /* ....*/ } // interdit !! c bien
public:
explicit toto(int u) { /* des trucs */ }
              toto(double u){/* des trucs */ }
};

int f(toto u);
toto e(5); // appel explicite du constructeur
int yy=f(3); // erreur pas d'appel implicite
yy = f(3.14); // ok le constructeur de toto a partir d'un double peut être
   appelé implicitement
```

Pourquoi tant de haine ? Bon tout les constructeurs de copie ne sont pas mauvais, mais en première lecture il faut mieux interdire qu'autoriser ! Si par hasard le constructeur de copie est une bonne idée alors nous le rendrons public mais seulement dans ce cas la !

Les pointeurs intelligents !

Commençons par le premier pointeur intelligent qui vous est fournit par la bibliothèque standard `auto_ptr`.

Ce pointeur permet de donner a un objet alloué sur le Tas le comportement de désallocation des objets alloués sur la pile.

```
void f()
{
Auto_ptr<char> p(new char[10000]); // hop j'ai pas besoin de faire delete
   en fin de fonction !
...
}
```

un autre avantage est le fait que si une exception est levée (ce qui n'est pas toujours catastrophique !) le tableau de char sera aussi libéré, ce qui n'est beaucoup plus compliquer a réaliser avec des appels explicites à delete (récupérer l'exception libérer le tableau puis relancer l'exception que l'on ne peut pas gérer a ce niveau).

Cette stratégie de l'objet de pile est bien utile pour réduire les fuites mémoire (ou plus grave libérer des verrous de fichier ou des mutex).

auto_ptr

Comment écrire `auto_ptr` ?
Nous allons l'écrire sous forme d'un template pour assurer un typage fort.
Premier jet cela donne :

```
template <class T>
class auto_ptr {
public:
auto_ptr(T *realptr = 0) : pointee(realptr) {}
~auto_ptr() { delete pointee; }
private:
T* pointee;
};
```

ok cool.

Cela veut dire que le auto_ptr est propriétaire de l'objet pointée que se passe t'il dans les cas de partage ? Ou d'affectation ? De copie ?

```
void f(){
auto_ptr<Truc> ptn1(new Truc);
auto_ptr<Truc> ptn2 = ptn1 ; // appel du constructeur de copy par défaut
   qui est propriétaire ? Que ce passe t'il ?
auto_ptr<Truc> ptn3;
ptn3 = ptn2 ; // oui et la ? operator =
```

} <- et bing trois appel de delete sur la même zone : "undefined behavior" dit la doc :(
en général c'est désastreux ....

auto_ptr (2)

Doit on faire des copies de la zone pointé ?
C'est délicat en effet on connaît le type de pointeur mais il peut pointer sur un tableau ou des objets de classe dérivées... on peut utiliser Clone() identifier le type avec type_info etc... LOURD surtout vu l'objectif de notre pointeur de pile.
Première solution (brutale) : interdire le constructeur de copie et l'opérateur d'affectation .

Bon , peut mieux faire, et c'est ce que fait auto_ptr il transfert la propriété des objets pointés au pointeur receveur.

```
template <class T>
class auto_ptr {
... // comme précédemment
public:
auto_ptr(auto_ptr<T> & par) { pointee = par.pointee; par.pointee = 0; }
auto_ptr<T> &operator=(auto_ptr<T> &par){
    if (this == &par) return *this; // auto ;) affectation
    delete pointee;          // libérer la propriété actuelle
    pointee = par.pointee; // prendre possession
    par.pointee = 0;         // se défaire de la propriété
    return *this;
}
};
```

=> ne pas faire d'appel par valeur avec un auto_ptr !
Pourquoi ? La propriété est transférée au nouveau pointeur sur la pile et disparaitra avec lui ce qui nous laisse avec un pointeur qui pointe sur 0.

auto_ptr

Vous avez biensur remarquer que nous avons utiliser des version non const des opérateurs d'affectation et de copie, en effet sur des versions const ces opérateurs sont interdit (étonnant non ?).

listting du fichier memory qui contient le code de auto_ptr.

auto_ptr