Chapitre VI

Patrons de fonctions et de classes

Sommaire

1 Introduction

Les modèles de classe (ou templates) sont un outils de substitution de code permettant de définir une classe dépendant de paramètres (typiquement types ou constantes), et d'instancier pendant la compilation ces paramètres pour les valeurs souhaitées.

Cela revient à disposer d'un code générique (par exemple, pour manipuler des vecteurs), et de demander au compilateur de générer les instances nécessaires pour le code écrit (par exemple, pour manipuler des vecteurs d'entiers, d'objet .

Exemple :

Pour calculer la somme des éléments d'un tableau `V` de `n` éléments de type `T`, le code est le suivant :

```
T sum(int n, T *V) {
    T sum = V[0];
    for(int i=1; i<n;i++) sum += V[i];
    return sum;
}
```

Puis, utiliser ce code générique avec `T=int` (ou `T=Vector`) pour effectuer des sommes de tableau d'entier (ou de `Vector`).

Note : Ceci suppose que les opérateurs =, += et la construction par copie existent pour le type `T`.

En `C`, ce type de fonctionnalité serait obtenu en utilisant des macros afin de générer du code.

2 Base

2.1 Principe

En C++, il est possible de créer 2 types d'objets génériques :

- des fonctions génériques,
- des classes génériques,

Principes

- un objet générique est défini en utilisant le mot-clef `template`, typiquement dans un fichier d'entête.
  Tel quel, le code générique est inactif.
- l'instance d'un objet générique (*i.e.* pour un type `T` donné) produit du code actif qui est compilé,

Conséquence : l'instanciation d'un objet générique génère de nombreuses lignes de code, et peut considérablement étendre la durée de la compilation.

2.2 Fonction générique

Syntaxe : `template <liste-paramètres> déclaration-fonction`

où `<liste-paramètres>` est une liste séparée par des virgules de paramètres utilisables dans la déclaration de la fonction parmi :

- `class T` : déclare un type générique `T` (alternative : `typename` équivalent `class` sauf pour les types dépendants).
- `int N` : déclare une entier `N` (cet entier doit pouvoir être explicité à la compilation).

`déclaration-fonction` est la déclaration de la fonction en utilisant le type générique `T` partout où le nom du type doit apparaître, et l'entier `N` partout où la valeur numérique doit être utilisée.

Exemple :

```
// définitions
template <class T> T Max(T x, T y) {
   return (x > y ? x : y);
}
template <class U, class V> U convert(V v) {
   return (U)v;
}
template <class V,int N> V *ArrayAlloc() {
   return new V[N];
}
```

La liste des paramètres est **explicitée** à la compilation (= remplacée) en fonction de l'instanciation choisie à l'appel de la fonction (*i.e.* type=nom du type utilisée, entier=valeur numérique).

L'appel d'une fonction générique s'effectue :

- soit en spécifiant la `<liste-arguments>` derrière le nom de la fonction (obligatoire pour les arguments entiers),
- soit en laissant au compilateur la déduction des types à partir des arguments passés (et non du type de retour), s'il n'y a pas d'ambigüité.

Exemple :

```
float a = Max<float>(3.4f,5.1f);   // idem Max(3.4f,5.1f)
float b = Max(6.1f,5); // échec: type T ambigü
int   c = Max<int>(6.1f,5);
int   d = Max<>(2,5);
int  *e = ArrayAlloc<int,100>();
```

Note : l'appel `Max(2,5)` est aussi possible, et a le même sens que `Max<>(2,5)` (voir plus loin les nuances sur ce point).

Remarques :

- par défaut, le code d'une fonction `template` n'est compilée que si elle est instanciée.
  conséquence : les erreurs qu'elle contient ne sont pas signalées à la compilation.
- **surcharge :** s'il existe une fonction de même nom et de mêmes paramètres, alors elle surcharge toute instance de fonction générique.
  Exemple : avec la surcharge `int Max(int x, int y) { ... }`
  - ◇ `Max(4,5)` appelle la surcharge fonctionnelle.
  - ◇ `Max<>(4,5)` appelle l'instance générique avec `T=int`.
- **paramètres par défaut :** il est possible de définir des valeurs par défauts dans la liste des paramètres.
  Exemple : `template <class V=int, int N=10> ...`
  Si le type (de retour) ou une valeur n'est pas spécifiée, alors la valeur par défaut est utilisée.

Il n'est pas possible de définir un pointeur sur une fonction générique (puisqu'elle n'est pas définie tant qu'elle est générique), mais cela devient possible dès qu'elle est instanciée :

- **déclaration d'un pointeur de fonction sur une instanciation d'une fonction générique :** cette déclaration provoque l'instanciation de la fonction générique pour créer le pointeur associé.
  Exemple :

```
int (*ptr)(int,int) = Max<int>;
int (*ptr)(int,int) = Max; // type déduit
```

  ceci déclare un pointeur `ptr` de type `int (*)(int,int)` sur l'instance de `Max<int>`.
- **passage du pointeur de l'instanciation d'une fonction générique :** le nom d'une fonction générique suivi de la liste des arguments permet de passer un pointeur sur l'instanciation de la fonction.
  Exemple :

```
int p(int x) { return x*x;  }
int g(int x, int(*pf)(int)) { return (*pf)(x); }
template <class T> T f(T x) { return x + 1; }
template <class T> T h(T x, T(*pf)(T)) { return (*pf)(x); }
```

```
int a= g(4,p);
int b= g(4,f<int>);
int c= h<int>(4, p);       // aussi avec h(4, p)
int d= h<int>(4, f<int>); // aussi avec h<int>(4, f)
```

Il est aussi possible de définir une fonction générique `inline`.

Syntaxe :

`template <liste-paramètres> inline déclaration-fonction;`

Exemple :

```
// définition
template <class T> inline T Max(T x, T y) {
   return (x > y ? x : y);
}
// utilisation identique
```

Rappel :

Il n'est pas possible d'utiliser un pointeur sur une fonction inline puisque celle-ci n'est jamais instanciée en tant que fonction (i.e. pas d'appel de fonction, mais remplacement du code au lieu de l'appel de la fonction).

2.3 Classe générique

Syntaxe : `template <liste-paramètres> déclaration-classe`

où `<liste-paramètres>` est une liste séparée par des virgules de paramètres utilisables dans la déclaration de la fonction parmi :

- `class T` : déclare un type générique `T` (alternative : `typename` équivalent `class` sauf pour les types dépendants).
- `int N` : déclare une entier `N`.

`déclaration-classe` est la déclaration de la classe en utilisant le type générique `T` partout où le nom du type doit apparaître, et l'entier `N` partout où la valeur numérique doit être utilisée.

Exemple :

```
template <class T, int N=10> class Stack {
    protected: T s[N];
    public: bool push(T x) {  /* code générique */ };
    ...
};
```

La liste des paramètres doit être **explicitée** à la compilation (= remplacée) en fonction de l'instanciation choisie à la définition de la classe (*i.e.* type=nom du type utilisée, entier=valeur numérique).

La définition d'un objet dont le type est une classe générique s'effectue en spécifiant la `<liste-arguments>` derrière le nom de la classe.

Un type instancié issue d'une classe générique peut être utilisé partout où nécessaire (définition d'objet, passage de paramètres, ...).

Exemple :

```
Stack<int>  s;    // tas de taille 10
s.push(4);
```

```
int fun(Stack<int> &s) { ... }
```

On sait que :

- La définition d'un objet générique s'effectue dans un fichier d'en-tête.
- La définition d'une classe complexe peut prendre de nombreuses lignes de code et dans ce cas, la définition de la classe doit également contenir le code générique de l'ensemble des méthodes.

Conséquence : il est nécessaire d'avoir un moyen de définir le code des méthodes génériques à l'extérieur de la définition de la classe.

Ceci s'effectue en général,

- en déclarant la classe générique avec les définitions des prototypes génériques des méthodes,

```
template <class T> class A {
  public: void f(T *x);
  ...
}
```

- en définissant à l'extérieur de la classe générique les méthodes avec l'opérateur de résolution de portée et en fournissant les arguments du modèle à la classe :

```
template <class T> void A<T>::f(T *x) { ... }
```

Exemple :

```
template <class T, int N>
class Stack {
protected:
    int n;
    T   s[N];
public:
    Stack() { n = 0; };
    bool pop(T&);
    bool push(const T&);
};
```

```
template <class T, int N>
bool Stack<T,N>::pop(T& v) {
    if (n==0) return false;
    v = s[--n];
    return true;
}

template <class T, int N>
bool Stack<T,N>::push(const T &x) {
    if (n == N) return false;
    s[n++] = x;
    return true;
}
```

```
Stack<int,10>    s;
s.push(4);
```

Les méthodes devant être implémentées afin que cette classe générique fonctionne pour un type `T` sont :

- un constructeur par défaut (allocation de `T[N]`),
- l'assignation par copie (pour le `push` et le `pop`)
- un déplacement par copie serait possible en modifiant légèrement le code dans le cas du `pop`.

Remarques :

- **surcharge :** une classe `template` ne peut pas être surchargée (nom unique), seulement être spécialisée (voir spécialisation).
  spécialisation = construction d'une variation de la classe générique en se basant sur sa définition.
- **paramètres par défaut :** il est possible de définir des valeurs par défauts dans la liste des paramètres.
  Exemple : `template <class V=int, int N=10> class ...`

Si le type (de retour) ou une valeur n'est pas spécifiée, alors la valeur par défaut est utilisée.

- **inline :** une méthode d'une classe template peut être `inline`.

3 Déduction de type

3.1 Cas général

Les règles de déduction de type sont l'ensemble des règles appliquées pour déterminer le type avec lequel un type générique sera appelé en fonction du type qui a été passé à l'objet générique.

Affirmation :

Soit le template : `template <class T> void f(T t);`

Soit l'appel : `f(p)` où `p` est de type `P`.

Alors il n'y a pas nécessairement `T = P`

Le type instanciée d'un objet générique suit donc des règles non triviales qu'il est nécessaire de connaître pour en comprendre le fonctionnement.

Il y a trois cas :

1. `P` est une référence ou un pointeur mais pas une référence universelle,
2. `P` est une référence universelle,
3. `P` n'est ni une référence ni un pointeur.

Cas 1 : si `T` est une référence (mais pas une référence universelle) ou un pointeur (rappel : `P` = type passé en paramètre)

Exemples : (avec éventuellement des modificateurs sur T)

```
template <class T> void f(T& p);
template <class T> void f(T* p);
```

Règles :

(a) si `P` est une référence, enlever la référence,

(b) faire du pattern matching pour déduire le type de `T` à partir de `P`

Application :

```
template <class T> void f(T& p)
template <class T> void g(const T& p)
int        x = 2;
const int  y = 3;
const int  &z = y;
f(x);  // P=int => T=int => appel f(int &p)
f(y); // P=const int => T=const int => appel f(const int &p)
f(z); // P=const int& -> const int => T = const int
      // => appel f(const int &p)
g(x); // P=int => T=int => appel g(const int& p)
g(y); // P=const int => T=int => appel g(const int &p)
g(z); // P=const int& => T=int => appel g(const int &p)
```

Cas 2 : si T est une référence universelle (rappel : P = type passé en paramètre)

Exemples : (avec éventuellement des modificateurs sur T)

```
template <class T> void f(T&& p)
```

Règles :

(a) si p est une lvalue, alors P et T sont des références à une lvalue.

(b) si p est une rvalue, appliquer les règles du cas 1.

Application :

```
template <class T> void f(T &&p)
int        x = 2;
const int  y = 3;
const int  &z = y;
f(x); // lvalue => P=int& et T=int& => appel f(int &p)
f(y); // lvalue => P=const int& et T=const int&
      // => appel f(const int &p)
f(z); // lvalue => P=const int& et T = const int&
      // => appel f(const int &p)
f(4); // rvalue => T=int et P=int => appel f(int &&p)
```

Note : parmi tous les cas, le cas 2(a) est le seul cas où on déduit que T est un référence et où P est déclaré comme la référence à une rvalue et est déduit comme la référence à une lvalue. Bref, c'est un hack afin de permettre le perfect forwarding.

Cas 3 : T n'est ni un pointeur ni une référence (rappel : P = type passé en paramètre)

dans ce cas, il s'agit d'un passage par valeur.

Exemples : (avec éventuellement des modificateurs sur T)

```
template <class T> void f(T p);
```

Règles :

(a) si P est une référence, enlever la référence.

(b) si P est `const`, alors ignorer `const`.

(c) si P est `volatile`, alors ignorer `volatile`.

Application :

```
template <class T> void f(T p)
int        x = 2;
const int  y = 3;
const int  &z = y;
f(x); // P=int => T=int => appel f(int p)
f(y); // P=const int -> const int  => T=int => appel f(int p)
f(z); // P=const int& -> const int -> int => T = int
      // => appel f(int p)
```

3.2 Cas du tableau statique

3.3 Cas particulier du tableau statique

Il s'agit du cas où l'argument passé est un tableau statique. Il est traité différemment du cas du pointeur.

Règles :

(a) **passage par valeur :** le tableau dégénère en pointeur sur le premier élément et la déduction est traitée comme telle.

(b) **passage par référence :** on obtient le type véritable du tableau (i.e. avec son nombre réel d'éléments).

Application :

```
const char name[] = "?";
template <class T> void f(T p);
template <class T> void g(T &p);
f(name); // T = const char*;
g(name); // T = véritable type du tableau = const char [2];
```

Note : il est possible de récupérer avec un template le type et la taille d'un tableau statique :

```
template<class T, std::size_t N> constexpr
   std::size_t arraySize(T (&)[N]) noexcept { return N; }
```

3.4 Cas particulier de la fonction

Il s'agit du cas où l'argument passé est une fonction (pointeur sur une fonction ou référence sur une fonction).

Règles :

(a) **passage par valeur :** déduction en pointeur sur une fonction.

(b) **passage par référence :** déduction en référence sur une fonction.

Application :

```
template <class T> void f(T p);
template <class T> void g(T &p);
void fun(int, double);
f(fun);  // T = void (*)(int,double)
g(fun);  // T = void (&)(int,double)
```

4 Spécialisation

Pour un objet générique, on entend par spécialisation d'un objet générique, la définition d'un objet de même nom (générique ou non), pour lequel on a :

- ajouté des modificateurs sur un type générique,
- défini des règles permettant de déduire l'instanciation d'un type générique à partir d'un autre,

- instancié directement un type générique par un type connu,

et ajouté un code dit spécialisé à cette définition.

La spécialisation d'un objet générique permet donc de disposer :

- d'une définition adaptée au type spécialisé, utilisée lorsque le type correspond à cette spécialisation,
- d'une définition générique générale utilisée dans tous les autres cas.

Une spécialisation d'un objet générique est une forme de **surcharge** de la définition générique.

On dit que :

- une **spécialisation** est **complète** ou explicite si elle explicite exactement tous les paramètres de l'objet générique : l'objet devient explicite.
- une **spécialisation** est **partielle** si elle réduit le champs des paramètres possible de l'objet générique, tout en laissant une part de généricité possible : l'objet reste générique.

4.1 Spécialisation complète

Une spécialisation explicite d'un objet générique se fait avec la syntaxe suivante :

`template <> déclaration-forcée { code-instancié }`

où :

- `déclaration-forcée` est la déclaration de l'objet où l'instance des paramètres a été forcée (*i.e.* comme dans une instanciation forcée),
- `code-instancié` est le code spécialisé de la fonction ou une méthode associée à l'objet générique.

Exemple :

```
struct Vec2 { float x,y; Vec2(u,v):x(u),y(v){}; };
// fonction générique
template <class T> T Max(T &u, T &v) {
    return (u > v ? u : v); }
// spécialisation complète T=vec2
template <> Vec2 Max<Vec2>(Vec2 &u, Vec2 &v) {
    return Vec2(Max<float>(u.x,v.x),Max<float>(u.y,v.y)); }
```

Une spécialisation explicite permet donc de donner le code spécialisé associé à un type particulier.

4.2 Spécialisation par modificateur

Une spécialisation par modification est une spécialisation partielle qui consiste à ajouter des modificateurs de type d'une façon qui limite les instanciations possibles.

Exemple :

```
// 1) fonction générique
template <class T> T Value(T x) { return x; };
// 2) spécialisation dans le cas où T est un pointeur
template <class T> T Value(T *x) { return *x; };
```

```
int a = 3, b;
b = Value<>(a);    // appel générique (1)
b = Value<>(&a);   // appel spécialisation (2)
```

Limitations : Toute spécialisation n'est pas possible car elle ne doit pas jamais conduire à une ambigüité sur le choix de la spécialisation à utiliser pour l'instanciation (voir ci-après).

Choix de la spécialisation : La fonction générique à utiliser parmi ses différentes spécialisations est celle qui spécialise le moins le type à instancier.

Exemples :

1) `template <class T> void f(T t)` (cas 3) enlève le const et la référence si présents dans le `t` passé. Donc, cette fonction :

   (a) est candidate pour les types `P`, `const P`, `P&` et `const P&`. Pas de spécialisation en aucun de ces types.

   (b) n'est pas candidate pour le type `P*`. Elle peut donc être spécialisée pour le type `T*` (attention, si P=int*, on reste dans le cas précédent).

2) `template <class T> void f(T &t)` (cas 2) enlève la référence si présents dans le `t` passé. Donc, cette fonction :

   (a) est candidate pour les types `P` et `P&`.

   (b) n'est pas candidate pour le type `const P&`, spécialisation possible (note : `const P` n'est pas une spécialisation de `P&`).

   (c) n'est pas candidate pour pour le type `P*` ; idem 1)(b) ci-dessus.

3) `template <class T> void f(T *t)` (cas 1) enlève la référence si présents dans le `t` passé. Donc, cette fonction :

   (a) est candidate pour les types `P*`, `P*&`. Pas de spécialisation pour `P*&`.

   (b) n'est pas candidate pour le type `const P*&`, spécialisation possible.

Principe derrière la déduction de type : (relire la section concernée pour les précisions) si la variable dans l'argument de l'objet générique est :

- indépendante de la variable passée, alors les qualificateurs sont perdus.
- dépendante de la variable passée, alors les qualificateurs sont conservés.

Exemples :

Spécialisations possibles en reprenant les spécialisations possibles de l'exemple précédent.

```
template <class T> void f(T t) { ... };
template <class T> void f(T* t) { ... };
template <class T> void f(const T* t) { ... };
```

```
template <class T> void f(T& t) { ... };
template <class T> void f(const T& t) { ... };
template <class T> void f(T* t) { ... };
template <class T> void f(const T* t) { ... };
```

4.3 Spécialisation partielle

Une spécialisation partielle d'un objet générique se fait avec la syntaxe suivante :

```
template <paramètres-partiels> déclaration-partielles {
  code-partiellement-instancié }
```

où :

- `paramètres-partiels` est la liste des paramètres restants à instancier,
- `déclaration-partielles` est la déclaration de l'objet contenant les `paramètres-partiels` et le reste des paramètres comme s'ils étaient tous instanciés.
- `code-partiellement-instancié` est le code partiellement spécialisé de la fonction/méthode associée à l'objet générique.

Exemple 1 :

```
// fonction générique
template <class U, class V> void fun(U u, V v) { ... };
// spécialisation partielle
template <class U> void fun<U,int>(U u, int v) { ... };
```

Exemple 2 :

```
// classe générique
template <class T1,class T2,int I> class A { ... };
// Spécialisation 1
template <class T,int I> class A<T,T*,I> { ... };
// Spécialisation 2
template <class T> class A<int,T*,5> { ... };
// Spécialisation 3 (par modificateur)
template <class T1,class T2,int I> class A<T1*,T2,I> { ... };
// Spécialisation 4 (par modificateur)
template <class T1,class T2,int I> class A<T1,T2*,I> { ... };
// Erreur: ceci n'est pas une spécialisation
template <class T1,class T2,int I> class A<T1,T2,I> { ... };
```

Règles de spécialisation partielle :

- la classe principale (= la plus générique) doit apparaitre avant toute spécialisation,
- une spécialisation partielle d'une classe est un template distinct, et une définition de tous les membres doit être fournie (≠ héritage).

La spécialisation partielle permet donc de répondre "progressivement" aux différents besoins de spécialisation des fonctions ou des classes.

5 Imbrications génériques

5.1 Héritage d'une classe générique

Exemple : classe générique de base utilisée

```
template <class T> class A {
protected: T a;
public: A(T v) : a(v) {};
    T get() const { return a; };
    void set(T &v);
};
template <class T> void A<T>::set(T &v) { a = v; }
```

Exemple : héritage d'une classe générique instanciée dans une classe

```
class B : public A<int> {
protected: int b;
public: B(int u, int v) : A<int>(u), b(v) {}
    int sum() const { return a + b; }
    int add();
};
int B::add() { return a + b; }
```

Notes :

- l'instanciation est précisée au moment de l'héritage,
- si besoin, les constructeurs font appels au constructeur instancié de la classe héritée.

Exemple : héritage d'une classe générique dans une classe

Cette fois, la classe générique héritée n'est pas instanciée.

```
template <class U> class C : public A<U> {
protected: int b;
public: C(U u, int v) : A<U>(u), b(v) {};
   U mul() { return A<U>::a * b; };
   U add();
};
template <class U> U C<U>::add() { return A<U>::a + b; }
```

Notes :

- la classe devient générique,
- utilisation du type générique dans la définition de la classe.
- si besoin, les constructeurs font appels au constructeur instancié de la classe héritée.
- les méthodes définies à l'extérieur de classe sont génériques.
- sur certains compilateurs, l'accès `A<U>::a` peut être remplacé par `a`.

Exemple : héritage d'une classe générique dans une classe générique

Cette fois, la classe générique héritée est héritée dans un classe générique.

```
template <class U, class V> class D : public A<U> {
protected: V b;
public: D(U u, V v) : A<U>(u), b(v) {};
   U add() { return A<U>::a + b; };
   U mul();
};
template <class U, class V> U D<U,V>::mul() {
    return A<U>::a * b; }
```

Notes :

- les paramètres hérités s'ajoutent aux paramètres de la classe générique,
- si besoin, les constructeurs font appels au constructeur instancié de la classe héritée.
- les méthodes définies à l'extérieur de classe sont génériques.

5.2 Méthode générique dans une classe

Exemple : Méthode générique dans une classe

On place des méthodes génériques (une définition interne, une déclaration externe) dans une classe non générique.

```
class E {
protected: int e;
public:
    E(int u) : e(u) {};
    template <class T> T add(T x) { return x + e; };
    template <class T> T mul(T x);
};
template <class T> T E::mul(T x) { return x*e;  }
```

Notes :

- les méthodes génériques sont définies à l'intérieur de la classe comme des fonctions génériques.
- les méthodes définies à l'extérieur de classe sont génériques.

Exemple : Méthode générique dans une classe générique

Les méthodes génériques (une définition interne, une déclaration externe) sont maintenant placées dans une classe générique.

```
template <class U> class F {
protected: U f;
public:
    F(U u) : f(u) {};
    template <class T> T add(T x) { return x + f; };
    template <class T> T mul(T x);
};
template <class U> template <class T>
    T F<U>::mul(T x) { return x*f; }
```

Notes :

- les méthodes génériques sont définies à l'intérieur de la classe comme des fonctions génériques.
- assez peu de différence avec le cas précédent si ce n'est dans le cas d'une définition externe pour lequel il faut faire un `template template`.
- **shadowing des paramètres d'un template :** de façon faire à ce que les noms de la liste des paramètres de la classe générique soient différents des noms de la liste des paramètres de la méthode générique.

Exemple : Méthode générique dans une classe générique qui hérite d'une classe générique

Les méthodes génériques (une définition interne, une déclaration externe) sont maintenant placées dans une classe générique.

```
template <class V, class U> class F : public E<V> {
protected: U f;
public:
    F(V v, U u) : E<V>(v), f(u) {};
    template <class T> T add(T x) { return x + f; };
    template <class T> T mul(T x);
};
template <class V, class U> template <class T>
    T F<U>::mul(T x) { return x*f; }
```

Notes :

- les paramètres hérités s'ajoutent aux paramètres de la classe générique,
- les méthodes génériques sont définies à l'intérieur de la classe comme des fonctions génériques.
- assez peu de différence avec le cas précédent si ce n'est dans le cas d'une définition externe pour lequel il faut faire un `template template`.

5.3 Friend générique

a) Amis dans une classe générique

L'utilisation du mot-clef `friend` est aussi possible dans un cadre générique :

- amis génériques dans une classe `A` non générique :

```
class A {
    // toute instance de la classe B (=B<T>) est amie de A
    template<class T> friend class B;
    // tout instance de la fonction f (=f<T>) est amie de A
    template<class T> friend void f(T);
};
```

- `friend` n'est pas autorisé sur la spécialisation partielle d'une classe :

```
template<class T> class A {};      // base
template<class T> class A<T*> {}; // spécial. partielle
template<> class A<int> {};        // spécial. totale
class X {
    template<class T> friend class A<T*>; // erreur
    friend class A<int>;                  // OK
};
```

Une relation d'amitié sur une classe générique se transmet aux membres d'une spécialisation totale dès lors qu'elle ne change pas leurs signatures.

- il est possible de donner la définition d'une fonction `friend` dans la déclaration d'amitié.

```
template class A { protected: int a;
public: friend int Fun1(A x) { return x.a; };
        template <class T> friend int Fun2(A x, T y)
           { return x.a + y; };
}
// Fun1 et le template Fun2<T> sont définis ici.
```

- les mots-clef `inline` et les arguments par défaut ne sont pas autorisés dans une déclaration `friend`, sauf si la définition est donnée avec la déclaration (et que les arguments par défaut ne sont pas template ou sinon explicitement typés).

```
class A { protected: int a;
public: template <class T> friend
          inline int Fun(A x, T y=0, int z=5)
            { return x.a + y + z; };
}
```

```
A a(4);
Fun(a,4,2);   // ok
Fun(a,4);     // ok = Fun(a,4,5)
Fun(a);       // erreur (détermination T impossible)
Fun<int>(a); // ok = Fun(a,0,5), T=int.
```

b) Amis génériques dans une classe générique

- La définition d'une fonction `friend` dans sa déclaration `friend` résout souvent directement les problèmes.
- **Première écriture :**

```
template <class T> class A {
  public: friend int Fun(A x);   // = Fun(A<T> x)
};
```

signifie que pour chaque type `T`, la fonction `int Fun(A<T> x)` est définie (noter que cette fonction n'est pas un template, car le template serait `int Fun<T>(A<T> x)`).
Exemple : si `Fun` est appelée avec un `A<int>`, alors la fonction `int Fun(A<int> x)` doit être définie. Donc, une définition manuelle et particulière par type (= non générique).

Sur certains compilateurs (g++), le warning "non-template function friend" est signalé pour cette écriture (l'écriture implique implicitement que `Fun` est générique, mais la déclaration ne le suppose pas).

- **Seconde écriture :** puisque la fonction est par essence générique, on la déclare comme une fonction générique :

```
template <class T> class A {
  public: template <class U> friend int Fun(D<U> x);
};
// définition de Fun (ou de toute spécialisation)
template <class U> int Fun(D<U> x) { ...  }
```

Ainsi, la fonction définie est `int Fun<T>(A<T> x)`. Elle génére un code pour l'ensemble des types nécessitant l'appel à `Fun`.

- **Troisième écriture :** classique si les prototypes des fonctions amies sont déjà connus au moment de la définition de la classe :

```
// déclaration préventive ou définition
template <class T> int Fun(D<T> x);
// utilisation
template <class T> class A {
  public: friend int Fun<>(A x);  };
```

Noter l'écriture `Fun<>(A x)` qui signifie que `Fun` est une instanciation de la fonction générique à partir du type déduit par `A` dans le contexte.
Cette écriture n'est pas possible s'il n'y a pas de déclaration préventive ou de définition avant la définition de la classe.

5.4 Virtuel générique

Dans le cadre de l'utilisation des templates, le mot-clef `virtual` ne fonctionne plus correctement.

A savoir (implémentation VS2015),

- la capacité à convertir vers la classe de base est perdue dès qu'il y a plus d'un héritage,
- l'héritage virtuel fonctionne, mais la fusion se fait sur la base nom de la classe + paramètres instanciation.
- le mot-clef virtuel n'est pas reconnu dès qu'une classe est générique.

Autrement dit, la virtualité classique n'est plus utilisable.

Voir la partie sur le CRTP comme alternative possible.

6 Résolution des dépendances

On définit :

- un **nom dépendant** est un nom dont la définition dépend de paramètres du template, et pour lequel il n'y a pas de déclaration dans la définition du template.
- un **nom non-dépendant** est un nom qui ne dépend pas des paramètres du `template`, le nom du template lui-même et ses noms déclarés internes (membres, amis, variables locales).

Le processus spécifié dans le standard pour chercher un nom dans un template est appelé "recherche de nom en deux phases" (two-phase name lookup) :

- **les noms dépendants** sont **résolus quand le template est instancié**, et peuvent nécessiter une désambiguïsation (=action pour lever l'ambigüité).
- **les noms non-dépendant** sont **résolus quand le template est défini** et ne nécessitent pas de désambiguïsation.

Important :

Dans la déclaration ou la définition d'un template, un nom dépendant qui n'est pas un membre de l'instanciation courante n'est pas considéré comme étant un type, sauf si le mot-clé `typename` est utilisé.

6.1 cas où typename est nécessaire

Donnons un exemple de cas où le mot-clé typename est nécessaire.

Exemple :

```
template<typename Container, typename T>
bool contains(const Container& c, const T& val) {
  Container::const_iterator
        iter = std::find(c.begin(), c.end(), val);
  return iter !=- c.end();
}
```

Trace de la compilation g++ (extraits) :

```
template.cpp: In function 'bool contains(const Container&, const T&)':
template.cpp:8:3: erreur: need 'typename' before 'Container::const_iterator'
    because 'Container' is a dependent scope
...
template.cpp:8:3: erreur: dependent-name 'Container::const_iterator'
    is parsed as a non-type, but instantiation yields a type
...
template.cpp:8:3: note: say 'typename Container::const_iterator'
    if a type is meant
```

Explication : le type Container::const_iterator est dépendant du type template Container, et en conséquence, il est obligatoire de le faire précéder du mot-clé typename, sinon il n'est interprété pas comme un nom de type.

Remarque : à noter que Visual Studio ne signale pas d'erreur. En conséquence, l'absence de typename rend votre code non portable.

6.2 Problèmes avec la recherche de nom

a) Exemple 1

Le premier exemple est un problème courant d'accès aux champs d'une classe de base template à partir d'une classe dérivée.

```
template<typename T> struct A {
  int a;
  void f() { ++a; }
};
```

```
template<typename T> struct B : A<T> {
  void g() { f();  return a; }
};
```

Trace de la compilation :

```
template.cpp: In member function 'void B<T>::g()':
template.cpp:8:15: erreur: there are no arguments to f that depend on a
    template parameter, so a declaration of f must be available
template.cpp:8:26: erreur: a was not declared in this scope
```

Explication :

1. Le champ a et la fonction f ne dépendent pas du paramètre template T (ce sont des noms non dépendants). Par ailleurs, A<T> dépend de T.

2. le compilateur ne cherche pas dans les classes de base ayant un nom dépendant lorsqu'il recherche des noms non dépendants. Donc, il ne connait ni `a` ni `f`.
   Ceci est seulement en lien avec la façon dont le compilateur recherche les noms. L'héritage fonctionne bien.

Solutions : plusieurs solutions différentes sont possibles :

1. utiliser l'écriture `this->` pour accéder aux membres de la classe mère.
   `this` est dépendant. En conséquence, la recherche est différée jusqu'au moment où le template est effectivement instancié. A ce moment, l'ensemble des classes de base sont considérées.

```
template<typename T> struct B : A<T> {
    void g() { this->f();  return this->a; }
};
```

2. utiliser l'écriture `using` afin de ramener les définitions des objets de la classe mère dans la classe dérivée.

```
template<typename T> struct B : A<T> {
    using A<T>::f;
    using A<T>::a;
    void g() { f();  return a; }
};
```

3. utiliser l'opérateur de résolution de portée. Cela rend le nom dépendant, et permet son inclusion dans la recherche.

```
template<typename T> struct B : A<T> {
    void g() { A<T>::f();  return A<T>::a; }
};
```

b) Exemple 2

Ce second exemple montre que, si la résolution de nom n'est pas comprise, l'assemblage des lien peut se faire de façon inattendue.

```
void g(double) { ... }
template<class T> struct S {
    void f() const {
        g(1); // g non dépendant: lien maintenant avec g(double)
    }
};
void g(int) { ... }
```

```
g(1);       // appelle g(int)
S<int> s;
s.f();      // appelle g(double)
```

Sortie :

- `g++ 4.9.2` : sortie comme ci-dessus.
- `VS2015` : sortie `g(int)` dans tous les cas (considère tous les contextes atteignables depuis l'instanciation).

Attention au compilateur utilisé :

faire en sorte que les fonctions appelées dans les templates soient bien toutes définies avant.

c) Exemple 3

Ce troisième exemple montre, qu'avec une résolution de nom mal maitrisée, une résolution qui devrait être locale est finalement effectuée à un niveau plus haut.

```
struct X { /*...*/ };    // global type (namespace scope)
void f() { /*...*/ }     // global function (namespace scope)
template <typename T> struct A {
  struct X { /*...*/ };
  void f() { /*...*/ }
};
template <typename T> struct B : A<T> {
  void g() {
    X    x;  // quel X est utilisé?
    f();     // quel f est utilisé?
  }
};
```

Que se passe-t-il ?

1. le type `X` et la fonction `f` dans `B` ne dépendent pas du paramètre template `T` (noms non dépendants). La classe de base `A<T>` a un nom dépendant.
2. le compilateur ne cherche pas dans la classe de base ayant un nom dépendant lorsqu'il recherche des noms non dépendants. Il ne connait pas le `X` et le `f` de `A<T>`, mais celui du namespace scope.

conséquence : le `X` et le `f` utilisés sont ceux du namespace scope, et non ceux de la structure de base template héritée.

6.3 Les deux à la fois

Ce dernier exemple montre qu'il est aussi facile d'écrire des codes qui conduisent aux deux problèmes à la fois.

```
template<typename T> struct A {
    struct X {  float t; };
    typedef int Y;
};
```

```
template<typename T> class B : public A<T> {
    public: void f() { X x;  Y y; }
};
```

Trace de la compilation :

```
template.cpp:9:4: erreur: 'X' was not declared in this scope
template.cpp:10:4: erreur: 'Y' was not declared in this scope
```

Explication :

1. les types `X` et `Y` ne dépendent pas du paramètre template `T` (ce sont des noms non dépendants). Par ailleurs `A<T>` dépend de `T`.
2. le compilateur ne cherche pas dans les classes de base ayant un nom dépendant lorsqu'il recherche des noms non dépendants. Donc, il ne connait pas `X` et `Y`.

Solution :

- utiliser l'opérateur de résolution de portée afin d'indiquer au compilateur où ces types se trouvent, à savoir `A<T>::`.
- inconvénient, les types `A<T>::X` et `A<T>::Y` écrit ainsi deviennent des noms dépendants. Il faut donc ajouter en plus `typename`.

```
// code corrigé
template<typename T> class B : public A<T> {
    public: void f() {
                typename A<T>::X   x;
                typename A<T>::Y   y;
            }
};
```

7 Outils

7.1 Alias (`using`)

Un alias `template` est un alias permettant de définir un nom équivalent à un template paramétré, qui permet d'utiliser l'alias directement comme s'il était lui-même un template.

Syntaxe :

```
template <liste-paramètres>
  using <alias-name> = <template-id>
```

où :

- `liste-paramètres` est la liste des paramètres du template.
- `template-id` est la déclaration d'un template préalablement défini utilisant la liste des paramètres, éventuellement avec spécialisation.
- `alias-name` est le nom utilisé pour l'alias.

Exemple :

```
template<class T> struct Alloc { ... };
template<class T> using Vec = vector<T, Alloc<T>>;
Vec<int> v; // = vector<int, Alloc<int>> v;
template<class T> void process(Vec<T>& v) { ... }
// idem process(vector<T, Alloc<T>>& v)
```

Note : évidemment, un alias ne peut pas faire référence à lui-même ; de manière directe ou indirecte.

7.2 `auto`

`auto x = expression;` : signifie que compilateur doit tenter de déterminer tout seul le type de la variable `x` à partir du type de l'`expression`.

Exemple 1 :

```
template<class T> void View(const vector<T>& v) {
   for (auto p = v.cbegin(); p != v.cend(); ++p) ...
}
```

Sinon, il aurait fallu écrire : `class vector<T>::const_iterator`.

Exemple 2 :

```
template<class T, class U>
  void multiply(const vector<T>& vt, const vector<U>& vu) {
    auto tmp = vt[i] * vu[i]; ...
}
```

Ici, le code n'est pas possible à écrire sans `auto` qui signifie = le type qui est obtenu lorsque l'on fait le produit d'un type `T` et d'une type `U`.

Règles

- le type résultant de `auto` peut être modifié (`const`, `static`, *, &, ...).
- `auto x = y;` si la variable `x` initialisée par `auto` est complètement indépendante de la variable `y`, alors `y` perd ses qualificateurs (`const`, ...).

7.3 `decltype`

Dans certains cas, le type de retour d'un template n'est pas connu :

Exemple :

Considérons une méthode dont le but est de retourner un type contenu dans un container par le biais d'un opérateur.

Le type retourné dépend alors du type retourné par la méthode définie dans le Container.

On utilise alors l'écriture :

```
template <class Container, class Index>
    auto Access(Container &c, Index i)
    -> decltype( c[i] ) {
    return c[i];
}
```

où l'on a :

- `auto` (voir leçon C_{11}^{++}) signifie que le type est automatiquement déterminé par le contexte, mais dans ce cas, il s'agit d'un type de retour qui doit donc être explicité,
- `decltype(x)` est une fonction qui retourne le type de son paramètre (voir plus loin).

Règle 1 :

`decltype(x)` retourne le type de définition de `x`, préserve la constance (i.e. si `x` est de type `const int`, alors `decltype(x)` est aussi de type `const int`).

Exemple :

```
struct Point { int x, y; };
Point p; // p.x est de type int
const Point& cp = p;
```

`decltype(cp.x)` ≡ `int` (type dans la définition du type).

`decltype((cp.x))` ≡ `const int&` (type dans la définition modifiée par le modificateur de la variable référente).

Règle 2 :

si l'expression d'une lvalue autre qu'un nom est de type `T`, alors decltype lui donne un type `T&`.

Exemple :

```
int x = 0;
```

`decltype(x)` ≡ `int` (type dans la définition du type).

`decltype((x))` ≡ `int&` (`(x)` n'est pas un nom, donc son type est `int&`).

L'intérêt de `decltype` dans les templates est donc de pouvoir retourner un type déduit qui ne peut être évalué qu'à partir de la connaissance d'un type générique ou interne.

8 Compilation

8.1 Instanciation forcée

Afin de forcer la compilation pour certains types, il est possible de déclarer les instances des fonctions, des classes ou des méthodes qui doivent être compilées.

Ceci est réalisé avec les écriture suivantes :

- **pour une fonction :** donner le prototype de la fonction, précédée du mot-clé `template`, avec tous les types/valeurs templates forcés.
  Exemple : `template int Max<int>(int,int);`
- **pour une classe :** donner la déclaration de la classe sans aucune déclaration interne, précédée du mot-clé `template`, avec tous les types/valeurs templates forcés.
  Exemple : `template class Stack<int,10>;`
- **pour une méthode :** idem fonction en utilisant l'opérateur de résolution de portée, où dans le nom de la classe, tous les types/valeurs templates sont forcés.
  Exemple : `template int Stack<int,10>::pop();`

8.2 Règles de compilation

Il est important de comprendre comment fonctionne la compilation des templates.

Règle 1 : Le code d'une fonction/classe `template` n'est compilé que s'il est instancié (*i.e.* explicitement utilisé) dans votre code.

Conséquence :

- lorsqu'un template est défini dans votre code mais pas instancié, alors son **code n'est jamais compilé**. Donc, s'il contient des erreurs (y compris de syntaxe), elles ne sont pas signalées à la compilation.
- Tout template défini doit être instancié afin de pouvoir être testé.
- Un template n'est compilé que pour les types que vous utilisez explicitement dans votre code.

Règle 2 : Tout objet doit être déclaré avec d'être utilisé (règle C++).

Conséquence : lorsqu'un template est utilisé pour un certain type :

- soit il est précédé par la définition complète du template
  ceci permet au compilateur de construire et de compiler le code nécessaire pour l'instanciation du template.
- soit il est précédé par la déclaration du template avec une spécialisation forcée pour le type utilisé.
  ceci permet d'indiquer au compilateur que le code compilé du template instancié est disponible ailleurs ; ce qui doit effectivement être fait afin d'éviter une erreur de lien.

Règle 3 : Tout objet déclaré `extern` dans une unité de traduction doit être défini dans une autre unité de traduction (règle C++).

Conséquence : pour un template, seule une spécialisation forcée peut être déclarée `extern`.

- Seules les spécialisations forcées déclarées sont utilisables.
- Ceci exige qu'il existe une autre unité de traduction contenant la déclaration des templates **ET** la spécialisation forcée afin de forcer leurs compilations.

Nous donnons maintenant des exemples de l'application de ces règles.

8.3 Template local

Un template est local s'il n'est disponible que dans l'unité de traduction dans laquelle il est utilisé.

Exemple :

```
// unit1.cpp
template <class T> T dummy(const T &t) { return t; }
// fonction utilisant le template
int fun(int x) { return x + dummy(4); }
```

Dans l'exemple ci-dessus, la fonction template `dummy` est instanciée pour le type `int` dans la fonction `fun`. Définie ainsi, seule l'instanciation `dummy<int>` est compilée, et elle n'est connue que de l'unité de traduction `unit1`.

8.4 Template classique

C'est la façon habituelle d'utiliser des templates : la définition des templates est placée dans un fichier d'en-tête `.h` qui est inclu dans toutes les unités de traduction qui utilisent sa définition.

Exemple :

```
// templ.h
template <class T> T dummy(const T &t) { return t; }
```

```
// unit1.cpp
#include "templ.h"
// fonction utilisant le template
int fun(int x) { return x + dummy(4); }
```

```
// unit2.cpp
#include "templ.h"
// fonction utilisant le template
float fun2(float x) { return x * dummy(2.f); }
```

Dans l'exemple ci-dessus, la fonction template `dummy` est instanciée pour le type `int` dans l'unité de traduction `unit1`, et pour le type `float` dans l'unité de traduction `unit2`. Le fichier d'en-tête permet de mettre ainsi à disposition des unités de traduction qui en ont besoin la définition des templates qui y sont définis.

8.5 Template externe

Dans certains cas, il peut sembler souhaitable de ne pas distribuer le code source d'un template, mais le mécanisme de compilation semble le rendre nécessaire.

Néanmoins, dans le cas où les types instanciés du template sont connus, il est possible d'utiliser la méthode suivante.

1. dans une unité de traduction, donner la définition du template ainsi que les instanciations forcées nécessaires.
2. dans un fichier d'en-tête, donner les instanciations forcées précédés du mot-clef `extern`.
3. inclure le fichier d'en-tête dans toutes les unités de traduction où les templates instanciés doivent être utilisés.

Le premier point permet de faire en sorte que le code des templates soit compilé pour les types attendus.

Exemple :

```
// templ.cpp
// définition
template <class T> T dummy(const T &t) { return t; }
// instanciations forcées dans l'unité de traduction
int dummy<int>(const int& t);
float dummy<float>(const float& t);
```

```
// templ.h : instanciations forcées externe
extern int dummy<int>(const int& t);
extern float dummy<float>(const float& t);
```

```
// unit1.cpp : utilisation
#include "templ.h"
// ok car défini dans templ.h et compilé dans templ.cpp
int fun(int x) { return x + dummy(4); }
```

```
// unit2.cpp : utilisation
#include "templ.h"
// ok car défini dans templ.h et compilé dans templ.cpp
float fun2(float x) { return x * dummy(2.f); }
```

Dans le cas d'une bibliothèque pour l'exemple ci-dessus, `templ.cpp` serait le contenu la bibliothèque

compilée (à partir duquel un .lib ou .dll est construit et avec lesquel on fait l'assemblage des liens), `templ.h` est le fichier d'en-tête de la bibliothèque, les autres des exemples d'utilisations.

8.6 Impacts sur la performance

Y-a-t-il un impact négatif des templates sur la vitesse ou l'occupation mémoire d'un programme ?

La réponse est non :

- Les templates sont seulement un outil de factorisation et de production de code.
- Une fois les variables templates instanciées, le code est compilé comme un code classique. Donc, il n'y a aucune perte de vitesse par rapport au même code écrit à la main, les spécialisations permettent les optimisations nécessaires pour les types souhaités.
- Mieux : les templates permettent des assemblages complexes de classe par assemblage/héritage de types/valeurs génériques, qui seraient longs et fastidieux de produire. mais ne réduit pas nécessairement le temps de développement car un code générique est plus long à écrire qu'un code classique.
- En terme d'occupation mémoire, seules les fonctions/méthodes utilisées sont compilées avec les types instanciés, donc pas plus qu'un code classique avec les types.

Néanmoins, la compilation d'un code template prend plus de temps, et d'autant plus s'il doit générer une quantité importante de code.

9 Contrôle de type

9.1 Gestion des instanciations

Limitation des types instanciées

En C++ classique, il est possible de limiter les types instanciés en déclarant un template mais en ne fournissant sa définition que pour certains types.

Exemple :

```
template <class T> T fun(T x);
template <> int fun<int>(int x) { return x*x; };

template <int n> struct ivect;
template <> struct ivect<1> { int x; };
template <> struct ivect<4> { int x, y, z, w; };
```

Les instanciations pour les types non définis engendrent des erreurs.

Interdiction d'instanciation

En interdisant certains types avec le mot-clef `=delete`.

Exemple :

```
template <class T> T square(T x) { return x*x; };
template <> int square<int>(int x) = delete;
```

Ces interdictions s'appliquent par extension à tous les types qui peuvent être convertis vers le type interdit (interdire `long int` interdit tout type entier car ils peuvent convertis en `long int`).

9.2 Constructions classiques

a) Type sous-jacent d'un conteneur

Un conteneur est une classe dont l'objet est de contenir des données. La méthode interne de stockage de données dans le conteneur est spécifique à celui-ci.

Les conteneurs sont très souvent des classes template car le conteneur s'occupe de la façon de stocker les données et non de ce qu'elles représentent.

Donc, supposons que nous avons un conteneur template :

```
template <typename T> class vector { ... }
```

et que nous passons un conteneur instancié pour un certain `T` à un autre template :

```
template <typename T> class A {
    // on veut ici le type interne stocké dans T
};
A<vector<int>>  a;
```

Question : comment connaître le type sous-jacent stocké dans le container ?
Dans l'exemple précédent, `T=vector<int>` pour la classe `A`, mais comment récupérer le type sous-jacent `int` dans `A` ?

Il y a trois solutions possibles.

Solution 1 :

ajouter un argument template supplémentaire qui contient le type sous-jacent.

```
template <typename S, typename T> class A {
    // ici : T=container S=type interne du container
};
A<int,vector<int>>  a;
```

Avantage : solution simple.

Inconvénients : ajoute un argument template supplémentaire, le type interne passé peut ne pas être cohérents avec le type interne du container.

Solution 2 :

utiliser le fait que les containers classiques (au moins ceux de la STL) définissent un type interne nommé `value_type`.

```
template <typename T> class A {
    using value_type = typename T::value_type;
    ...
};
A<vector<int>>  a;
// pour cet instanciation, value_type = int
```

Avantage : solution simple pour les containers

Inconvénients :

- suppose que le container implémente le type interne `value_type`
  Note : toujours prendre exemple sur la STL pour implémenter vos propres container car vous apprend de bons réflexes.
- suppose que le template ne prend que des containers en paramètre.
  ceci peut être assuré avec les traits de type (voir plus loin)

Solution 3 :

utiliser un proxy pour séparer le type de container et le type stocké dans le container.

```
// proxy pour le container contenant le type
template<typename T> struct wrap_vector {
  using type = std::vector<T>;
};

// définir le container interne à partir du wrapper et du type
template <template<class> class T,class S> struct A {
    typename T<S>::type   data;
};
// définition séparant le container et le type
A<wrap_vector,int>  a;
```

Avantage : solution élégante

Inconvénient : oblige à définir des wrappers spécifiques pour tous les containers utilisés.

Solution 4 :

utiliser les template de template.

```
// définir le container interne à partir du wrapper et du type
template <template<class,class> class T,class S> struct A {
    T<S,std::allocator<S>>   data;
};

// définition séparant le container et le type
A<std::vector,int>  a;
```

Avantage : solution interne au langage.

Inconvénient : oblige le template de template à avoir exactement le même nombre d'arguments que le template passé en paramètre (la valeur des arguments par défaut n'est pas prise en compte).
Dans l'exemple ci-dessus, oblige le second argument par défaut à être un allocateur.

b) Politique

On définit :

- une **politique** est une interface de classe (éventuellement générique),
- une **politique de classe** est l'implémentation (=une réalisation) d'une politique.
- une **classe hôte** est une classe qui utilise une politique de classe.

Utilisation :

- Les politiques servent à gérer les aspects fonctionnels, et permettent aussi de paramétrer les types.
- Étant basé sur le template, le résultat pourra être optimisé par le compilateur.

Construction :

- On utilise l'héritage générique pour hériter de la politique que l'on souhaite intégrer à la classe.
- La situation est inversée par rapport à l'héritage (héritage : la classe mère est spécialisée par la classe courante, politique : la classe courante est spécialisée par la classe mère).
  i.e. la relation d'héritage n'a plus le sens courant.

Exemple : première approche d'une politique

```
// politique externe
class Policy1 {
    public: alias type = int;
    static type f() { return type(3); };
}
class Policy2 {
    public: alias type = long int;
    static type f() { return type(15L); };
}
```

```
// classe hote
template <class Policy> class Host : public Policy {
public:
    // modification d'une méthode interne par une
    // politique externe
    typename Policy::type g() {
        return typename Policy::type(5); }
    // la politique associé à f() est héritée
};
```

```
Host<Policy1>  h;
h.f();    // f issue de la politique P1
h.g();    // type de g issue de la politique P1
```

Exemple : classe hôte template héritant d'une politique template

```
// politique externe
template <class T> class Policy1 {
    public: alias type = typename T;
    static type f() { return type(5); };
}
```

```
// classe hote
template <class T, template <class I> class Policy>
class Host : public Policy<T> {
public:
    typename Policy<T>::type g() {
        return typename Policy<T>::type(5) / 2; }
    // la politique associé à f() est héritée
};
```

```
Host<float,Policy1>  h1;
h1.f();   // f issue de la politique <float,P1>
h1.g();   // type de g issue de la politique <float,P1>
Host<int,Policy1>    h2;
h2.f();   // f issue de la politique <int,P1>
h2.g();   // type de g issue de la politique <int,P1>
```

c) CRTP

Principe : CRTP (Curiously Recurring Template Pattern) désigne un motif d'héritage dans lequel une classe template de base reçoit comme valeur pour son paramètre template une de ses classes dérivées :

- cela revient à dire qu'à l'instanciation la classe de base peut connaître ses sous-classes
- cette technique permet d'obtenir le même effet que des méthodes virtuelles, sans le coût associé (static polymorphism)
- elle peut être utilisée pour obtenir une notion de conformité vis-à-vis d'une interface avec génération de code et un typage correct

Exemple : CRTP

```
template <class T> class BASE {
public:
  void interface() { static_cast<T *>(this)->impl(); }
};
```

```
class DERIVE1 : public BASE<DERIVE1> {
public:
  void impl() { cout << "DERIVE1::impl()" << endl; };
};
```

```
class DERIVE2: public BASE<DERIVE2> {
public:
  void impl() { cout << "DERIVE2::impl()" << endl; };
};
```

```
// utilisation directe
DERIVE1 d1;
d1.interface(); // appel « dynamique »
DERIVE2 d2;
d2.interface(); // appel « dynamique »
```

L'interface est bien définie dans la classe mère et implémentée dans les classes filles.

Explication :

- la classe parent (générique) se voit injecter à l'instanciation (au sens de la génération par le compilateur de la classe à laquelle s'applique le template) le type de son enfant.
- cette technique est possible parce que le nom de l'enfant apparaît avant le nom du parent et existe donc au moment où le parent est généré par le compilateur
- ainsi la sélection de la méthode à appeler est réalisée statiquement (à la compilation), il n'y a donc pas de surcoût lié au dynamisme (static polymorphism)

Dans l'exemple précédent, les objets obtenus ne sont évidemment pas polymorphiques, puisque :

- les classes qui dérivent de `BASE` n'héritent pas de la même version de `BASE` (le type `T` diffère),
- en conséquence, avec cette définition, `DERIVE1` et `DERIVE2` n'ont pas la même classe mère, et ne représentent donc pas deux formes différentes d'une même `BASE`.

Essai 2 : CRTP avec héritage simple

```
class Vehicle {
public:
  virtual ~Vehicle() {}
  virtual Vehicle *clone() const = 0;
  virtual void describe() const = 0;
};
```

```
template <typename Derived>
class VehicleCloneable : public Vehicle {
public:
  virtual Vehicle *clone() const {
    return new Derived(static_cast<Derived const &>(*this));
  }
};
```

```
class Car : public VehicleCloneable<Car> {
public:
  virtual void describe() const { cout << "car" << endl; }
};
```

```
class Plane : public VehicleCloneable<Plane> { ... };
```

Avec cette implémentation :

- on ajoute une classe mère dont hérite le CRTP afin d'avoir une classe de base,
- `clone` reste une méthode statique pour `Car`, mais est une méthode virtuelle pour `Vehicle`.

Problème : si on veut définir maintenant une classe qui hérite de `Car`,

- ```
  class Tank : public Car { ... };
  ```
  la méthode `clone` ne fonctionne plus correctement (c'est celle de `Car`).
- ```
  class Tank
      : public VehicleCloneable<Tank> { ... };
  ```

la classe n'hérite plus de `Car`.

- ```
  class Tank : public VehicleCloneable<Car> { ... };
  ```
  on a les deux problèmes en même temps.

Il faut donc modifier le CRTP pour être en mesure de faire de l'héritage multi-niveaux.

Essai 3 : CRTP avec héritage multi-niveaux

```
template <typename Base, typename Derived>
class VehicleCloneable : public Base {
public:
  virtual Base *clone() const {
    return new Derived(static_cast<Derived const &>(*this));
  }
};
```

```
class Car : public VehicleCloneable<Vehicle, Car> { ... };
```

```
class Tank:
  public VehicleCloneable<Car,Tank> {
public:
  virtual void describe() const { cout << "Tank" << endl; }
};
```

```
Vehicle  *volt = new Tank();
volt->describe();     // "Tank"
Vehicle  *other = volt->clone();
other->describe();    // "Tank"
```

Problèmes avec la solution : si on veut ajouter des constructeurs de la manière suivante :

Exemple :

```
class Vehicle {
  protected: int Fuel;
  public: Vehicle(int fuel) : Fuel(fuel) {}
  ... };
```

```
class Car : public VehicleCloneable<Vehicle, Car> {
  private: typedef VehicleCloneable<Vehicle, Car> BaseClass;
  protected: int Doors;
  public: Car(int fuel, int doors)
            :BaseClass(fuel),Doors(doors) {}
  ... };
```

```
class Tank : public VehicleCloneable<Car,Tank> {
  private: typedef VehicleCloneable<Car,Tank> BaseClass;
  protected: int Guns;
  public: Tank(int fuel, int doors, int guns)
             :BaseClass(fuel, doors), Guns(guns) {}
  ... };
```

Malheureusement, ce code ne compile pas : dans `Car`, la classe de base est `VehicleCloneable<Vehicle, Car>` et non `Vehicle`. En conséquence, l'appel au constructeur de `Vehicle` échoue.

Il faut trouver un moyen de ramener les constructeurs de la classe de base dans `VehicleCloneable`. En C^{++}_{11}, la solution est la suivante :
```

```
template <typename Base, typename Derived>
class VehicleCloneable : public Base {
public: using Base::Base;
        virtual Base *clone() const { /* idem */ }
};
```

où l'écriture `using Base::Base` permet de faire en sorte que tous les constructeurs de `Base` d'être importé dans `VehicleCloneable` (admirez : Base est un argument template). Cette modification de `VehicleCloneable` rend compilable et valide le code précédent.

En C++ antérieur, la seule solution consiste à forwarder les constructeurs, c'est à dire à répéter ceux de la classe de base et à les appeler explicitement, à savoir :

```
template <typename Base, typename Derived>
class VehicleCloneable : public Base {
public:
    VehicleCloneable(int arg1) : Base(arg1) {}
    // à répéter pour toute combinaison d'arguments possibles
    // sur l'argument template Base.
    virtual Base *clone() const { /* idem */ }
};
```

9.3 Traits d'un type

Les deux approches précédentes ont pour inconvénient d'avoir à citer explicitement tous les types autorisés ou interdits.

Les traits d'un type sont un ensemble de procédés permettant de déterminer des propriétés sur les types génériques.

Dans notre contexte, le mot trait est pris au sens de caractéristique ou de signe distinctif (exemple : traits d'un visage).

Définition de Bjarne Stroustrup :

> penser au trait comme un petit objet dont le but principal est de transporter de l'information utilisé par un autre objet ou algorithme afin de déterminer une politique ou des détails d'implémentation.

Le trait d'un type est implémenté à travers une méta-fonction.

Une méta-fonction est une fonction générique qui prend en paramètre un ou plusieurs types et retourne des types ou des constants.

Exemple : méta-fonction vérifiant si deux types sont égaux

```
// définition du concept T1==T2
template <class T1, class T2> struct typeequal {
   static const bool value = false;                  };
template <class T> struct typeequal<T,T> {
   static const bool value = true;           };
```

```
// utilisation
bool bEq1 = typeequal<int,char>::value;   // faux
bool bEq2 = typeequal<int,int>::value;    // vrai
```

Principes utilisés :

- une classe générique contient le résultat value par défaut (=false) pour deux types quelconques,
- une spécialisation dans laquelle les types sont identiques change le résultat pour cette spécialisation.
- à la compilation, une variable statique `typeequal<T1,T2>::value` est créée pour tout couple `T1` $\neq$ `T2` instancié ; idem pour `typeequal<T>::value` dans le cas où `T1=T2=T`.

Exemple : méta-fonction transformant un type

```
// cas général
template <typename T>
   struct remove_pointer { typedef type = T; };
// cas où T est un pointeur
template <typename T>
   struct remove_pointer<T*> { typedef type = T; };
```

```
// utilisation
typename remove_pointer<int>::type    a;  // a type int
typename remove_pointer<int*>::type   b;  // b type int
```

Amélioration :

```
// définition utilisant les déclaration de remove_pointer
template <typename W> using RemovePointer
      = typename remove_pointer<W>::type;
// utilisation moins verbeuse
RemovePointer<int>   a2;
RemovePointer<int*>  b2;
```

Exemple : exemple fonctionnel de transformation de type

```
template <typename T> struct CallTraits {
  // implémentation générale
  template <typename U, bool Big> struct Impl;
  // spécialisation pour Big=true: Type = U&
  template <typename U> struct Impl<U, true> {
      typedef const U& Type;                          };
  // spécialisation pour Big=false: Type = U
  template <typename U> struct Impl<U, false> {
      typedef U Type;                                  };
  // ParamType<T> avec Big = sizeof(T)>8
  using ParamType
     = typename Impl<T,(sizeof(T)>8)>::Type;
};
```

```
// utilisation
template <typename T> T Min(
    typename CallTraits<T>::ParamType X,
    typename CallTraits<T>::ParamType Y) {
    return X < Y ? X : Y;
}
```

```
// passage par référence constante
std::string s3 = Min<std::string>(s1, s2);
// passage par valeur
int i3 = Min<int>(i1, i2);
```

Depuis C_{11}^{++}, de très nombreux traits ont été intégrés (ils étaient dans `boost` avant), principalement :

- **les traits conditionnels** servent à tester si un type vérifie une certaine propriété.
  Exemples :
  - ◇ `is_integral<T>` vérifie si le type `T` représente un type entier,
  - ◇ `is_default_constructible<T>` vérifie si le type `T` possède un constructeur par défaut,
  - ◇ `is_const<T>` vérifie si le type `T` est `const`.
  - ◇ ...
- **les traits modificateurs de type** servent à transformer un type en un autre (construit à partir du premier).
  Exemples :
  - ◇ `remove_const<T>` qui retire le qualificateur `const` s'il est présent dans `T`,
  - ◇ `add_pointer<T>` qui transforme le type `T` en `T*`,
  - ◇ ...
- **les traits transformateurs de type :** servent à transformer un type en un type différent. Parmi ceux-ci, le plus utilisé est **`enable_if`** qui permet de n'autoriser l'instanciation d'un template si une condition particulière est vérifiée (SFINAE).

a) Traits conditionnels

Les traits conditionnels sont les suivants :

- **Catégorie d'un type :** `is_x` avec **x** = `void`, `null_pointer`, `array`, `pointer`, `enum`, `union`, `class`, `function`, `object`, `scalar`, `compound`, `integral`, `floating_point`, `fundamental`, `arithmetic`, `reference`, `lvalue_reference`, `rvalue_reference`, `member_pointer`, `member_object_pointer`,`member_function_pointer`.
- **Propriétés d'un type :** `is_x` avec **x**= `const`, `volatile`, `pod`, `empty`, `polymorphic`, `final`, `abstract`, `trivial`, `trivially_copyable`, `standard_layout`, `literal_type`, `signed`, `unsigned`.
- **`Opérations supportés:`** `is_x, is_trivially_x`, `is_nothrow_x` avec `x=constructible`, `default_constructible`, `copy_constructible`, `move_constructible`, `assignable`, `copy_assignable`, `move_assignable`, `destructible` et `has_virtual_destructor`.
- **Propriété des relations :** `is_same`, `is_base_of`, `is_convertible`, `alignment_of`, `rank`, `extent`.

Un trait conditionnel `is_xxx` exposent essentiellement :

- la valeur du résultat à travers un champ statique nommé `value`

- à partir du C_{14}^{++}, ce résultat sera également accessible avec l'opérateur `()` où l'équivalent `{}` (noter que cette écriture peut échouer dans certains contextes)
- à partir du C_{17}^{++}, la valeur du résultat sera accessible directement avec l'alias `is_xxx_v`

Exemple : `is_integral`

```
// les trois écritures ci-dessous sont équivalentes
bool v1 = is_integral<int>::value;  // true
bool v2 = is_integral<int>();       // à partir de C++14
bool v3 = is_integral<int>{};       // écriture équivalente
bool v4 = is_integral_v<int>;       // à partir de C++17
```

Remarques :

Seule la dernière écriture n'est actuellement pas disponible sous VS2015 ou g++ 4.9.2.

Elle peut être facilement définie avec :

```
template <class T>
    using is_integral_v = typename is_integral<T>::value;
```

b) Traits modificateurs de type

Les traits modificateurs de type sont les suivants :

- `remove_x` où **x** = `cv`, `const`, `volatile`, `reference`, `pointer`, `extent`, `all_extents`.
- `add_x` où **x** = `cv`, `const`, `volatile`, `pointer`, `lvalue_reference`, `rvalue_reference`.
- `make_x` où **x** = `signed`, `unsigned`.

Les traits modificateurs exposent essentiellement :

- la type modifié est exposé dans un `typedef` public nommé `type`.
- à partir du C_{14}^{++}, la type du modificateur `xxx` sera accessible directement avec l'alias `xxx_t`.

Exemple : `add_const`

```
template <class T> T myfun(add_const_t<T> x)
    {  /* ici x est const T */  }
int v = myfun<int>(10);
add_const<int>::type w=5;
```

c) Traits transformateurs de type

Les transformateurs de type ont pour but de transformer un type en un autre différent.

Transformation de type :

`aligned_storage`, `aligned_union`, `decay`, `enable_if`, `void_t`, `conditional`, `common_type`, `underlying_type`, `result_of`.

Exemples : (parmi les plus utiles)

- `decay` : renvoit le type dégénéré associé (exemple : lvalue vers rvalue, tableau vers pointeur, fonction vers pointeur)
- `result_of` : déduit le type d'une expression.
- `enable_if` : permet de définir une instanciation conditionnelle du template (voir SFINAE).

Donnons maintenant des détails sur `enable_if` qui est de loin la plus utilisée.

d) `enable_if`

`enable_if` permet de transformer un booléen en un type `T` sous la condition qu'une condition `B` soit vérifiée. Dans le cas contraire, l'instanciation du type est désactivée.

Ce template a le prototype suivant :

```
template<bool B, class T=void> struct enable_if;
```

Typiquement l'utilisation est la suivante :

- la place de `B` est occupée par une valeur booléenne, par exemple la valeur d'un trait conditionnel,
- `enable_if` ne sait renvoyer qu'un type, et s'il n'est pas précisé, ce type est `void` par défaut.
- si la condition définie dans `B` est vraie, alors `enable_if` retourne `T`, sinon il provoque l'échec de l'instanciation du type (voir SFINAE).

`enable_if` ne peut retourner que son type interne nommé `type` :

- avec l'opérateur de résolution de portée : `enable_if<B,T>::type`
- avec l'alias équivalent (depuis C_{14}^{++}), `enable_if_t<B,T>`.

Le détail de l'utilisation de `enable_if` en pratique est donné dans la section suivante.

9.4 SFINAE

Principe :

- SFINAE (Substitution Failure Is Not An Error) signifie que lorsque le compilateur substitue les types dans la déclaration d'une fonction template, et que cette substitution échoue, ceci ne constitue pas une erreur
- le compilateur continue en recherchant d'autres templates, jusqu'à ce qu'une substitution soit possible (si finalement il n'y en a pas alors cela devient une erreur)
- autrement dit, si la substitution des types dans une fonction template échoue, ce template est discrètement éliminé du jeu, sans signalement d'erreur

Le trait `enable_if<B,T>` a été conçu pour fonctionner dans ce cadre : si la condition `B` est vérifié, le type `T` est instancié, sinon l'échec de l'instanciation ne provoque pas d'erreur, et conduit le compilateur à en chercher une autre plus appropriée.

Concrètement, nous sommes dans l'un des cas suivants :

- on a une fonction template :

```
template <class T> A funA(...);
```

- on a une classe template :

```
template <class T> class A { ... };
```

et l'on souhaite définir une condition sur `T` afin de restreindre les types sur lesquels peuvent s'instancier ces templates.

Cette restriction peut être effectuée de trois manières différentes :

1. dans le type de retour d'une fonction
2. dans le paramètre d'une fonction
3. dans le paramètre d'un template de classe ou de fonction.

Ces méthodes sont présentées dans le cadre de `enable_if`, mais on pourra s'inspirer de ces méthodes pour tout trait ne l'utilisant pas.

a) SFINAE dans le type de retour d'une fonction

Pour utiliser le type de retour d'une fonction afin de tester la condition `B`, il faudra donc `enable_if` retourne le type souhaité (ici `A`) :

```
// fonction template initiale sans contrainte sur T
template <class T> A funA(...);
// idem (exclusif) en testant T dans le type de retour
template <class T>
    typename enable_if<...,A>::type fun(...) { ... }
```

Exemple : soit la fonction template suivante :

```
template <class T> T incr(T x) { return x + T(1); }
```

type de retour modifié afin de restreindre T aux types entiers

```
template <class T>
    // ligne suivante = le type de retour (ici T)
    typename enable_if<is_integral<T>::value,T>::type
      incr(T x) { return x + T(1); }
```

Noter que :

- cela nuit à la lisibilité du type de retour de la fonction, donc à réserver au cas où `A=void` (la valeur par défaut du type pour `enable_if`).
- `typename` est nécessaire car la définition du type de retour dépend de `T`.
- si le type de retour de la fonction est `void`, le type devient optionnel (i.e. utiliser `enable_if<B>::type` au lieu de `enable_if<B,void>::type`)
- en C^{++}_{14}, utiliser `enable_if_t<...>`.

b) SFINAE dans le paramètre d'une fonction

Toujours dans le cadre d'une fonction template, on voudrait tester maintenant le type du template dans l'argument de la fonction :

problème : si l'on utilise l'un des paramètres quelconque de la fonction, on va fortement nuire à la lisibilité de la liste des paramètres

solution : utiliser les paramètres optionnels du C^{++} en ajoutant un dernier paramètre inutile non nommé ayant une valeur par défaut.

Le `enable_if` renvoyant un `void`, le plus simple est d'ajouter une étoile afin de pouvoir initialiser la valeur par défaut de ce dernier paramètre (`void*`) à 0 (i.e. équivalent à écrire `void*=0` si vrai) comme suit :

```
// fonction template initiale sans contrainte sur T
template <class T> A funA(...);
// idem (exclusif) en testant T en dernier argument
template <class T>
  A fun(..., typename enable_if<...>::type* = 0) { ... }
```

Exemple : soit la fonction template suivante :

```
template <class T> T incr(T x) { return x + T(1); }
```

paramètre optionnel ajouté restreignant T aux types entiers

```
template <class T> T incr(T x,
  typename enable_if<is_integral<T>::value>::type* = 0)
    { return x + T(1);  }
```

c) SFINAE dans le paramètre d'un template

Une technique similaire à celle des paramètre d'une fonction peut être utilisée basée sur le fait que les paramètres d'un template peuvent aussi avoir des valeurs par défaut.

Le `enable_if` renvoyant un type, on peut utiliser ce type comme étant un type par défaut du paramètre non nommé ajouté au template comme suit :

```
// fonction template initiale sans contrainte sur T
template <class T> A funA(...);
// idem avec argument du template:
template <class T, typename = enable_if<...>>
```

Exemple : soit la fonction template suivante :

```
template <class T> T incr(T x) { return x + T(1); }
```

paramètre optionnel ajouté restreignant T aux types entiers

```
template <class T,
  typename = enable_if<is_integral<T>::value>>
    T incr(T x) { return x + T(1); }
```

Notes :

- le paramètre du template n'a pas besoin d'être nommé si son résultat n'est pas utilisé. Sinon, il est possible de renvoyer un vrai type et/ou une vrai valeur comme argument du template s'il en a l'usage.
- des écritures similaires aux méthodes précédentes peuvent aussi être utilisée, mais elles sont moins compactes.

10 Référence universelle

10.1 Problèmes avec les déplacements

Les références à une rvalue devraient inconditionellement être castées en rvalues (avec std : :move) lorsqu'elles sont transmises à d'autres fonctions, car elles sont toujours associées à des rvalues.

Donc, la manière standard d'implémenter un constructeur par déplacement devrait être :

```
class A { public: A(A&& rhs):s(std::move(rhs.s)) {} };
```

Le `move` est **obligatoire** car sinon rhs est une `lvalue`.

Si on veut implémenter un setter qui fixe ce même champ, il faudrait écrire un code qui gère séparément la rvalue et la lvalue :

```
class A { public:
void setS(const S& new_s) { s = new_s; }       // APC call
void setS(S&& new_s) { s = std::move(new_s); } // APD call
```

= deux codes à écrire et à maintenir.

Pire, si la fonction a maintenant *n* paramètres qui peuvent être des `lvalue`s ou des `rvalue`s, il faudrait écrire 2^n fonctions pour gérer toutes les combinaisons de `lvalue`s et de `rvalue`s.

Pour résoudre ce problème, on utilise les templates avec les références universelles.

10.2 Fusion des références

Avant C_{11}^{++}, il n'était pas autorisé de prendre la référence d'une référence (quel sens donner à cela ? Le compilateur considère l'écriture `A& &a` malformée).

A partir de C_{11}^{++}, il y a deux types de référence (sur une `lvalue` et sur une `rvalue`), et une sémantique particulière a été donnée au référence de référence (connu sous le nom de fusion de référence).

La fusion des références a lieu lorsque, dans un template ou un `auto`, l'écriture conduit à une référence de référence :

Type	Argument	Résultat
`T&`	`&x`	`T&`
`T&`	`&&x`	`T&`
`T&&`	`&x`	`T&`
`T&&`	`&&x`	`T&&`

Exemple:

```
typedef int&  lref;
typedef int&& rref;
int n;
lref&  r1 = n; // type of r1 is int&
lref&& r2 = n; // type of r2 is int&
rref&  r3 = n; // type of r3 is int&
rref&& r4 = 1; // type of r4 is int&&
```

10.3 Référence universelle

Une référence universelle est une référence de type `T&&` qui peut être interprétée comme `T&` ou `T&&`, à savoir :

- si l'expression initialisant la référence universelle est une `lvalue` : la référence universelle devient une référence à une `lvalue`,
- si l'expression initialisant la référence universelle est une `rvalue` : la référence universelle devient une référence à une `rvalue`.

Comment définir une référence universelle ?

Une référence universelle est une référence de type T&& uniquement si le type T peut être déduit à partir du contexte de l'appel, à savoir :

- soit une variable de type `auto &&x`,
- soit une variable dans un template : `template <typename T> void f(T&& x)`

Exemples :

- `template <typename T> void f(int&& x)`
  x n'est pas une référence universelle.
- `template<typename T> void f(std::vector<T>&& x)`
  n'est pas une référence universelle (car paramètre de type `std::vector<T>&&` et non T&&).
- `template<class T> class A { ... public: void f(T &&x); ... };`
  n'est pas une référence, car l'instanciation de T a lieu avant d'avoir besoin de la fonction f.

Attention :

- la réduction de type est nécessaire,
- le modificateur `const` disqualifie une référence d'être universelle

Utilisons maintenant la référence universelle pour tenter de résoudre notre problème de setter :

Exemple :

```
class A {
private: std::string s;
public:
  template<typename T> void setS(T&& new_s) {
    // new_s référence universelle
    // = de type T&& si rvalue et T& si lvalue
    s = std::move(new_s);
  }
};
```

```
A a;
a.setS("tutu");   // rvalue : ok, pas d'objet temporaire
auto n = std::string("toto");
a.setS(n);        // lvalue : valeur de n inconnue
```

Problème : maintenant, le move étant inconditionnel, il déplace aussi les `lvalue`s.

Il faudrait un moyen d'effectuer un `move` si l'argument du template est une `rvalue`, et un passage classique si l'argument est une `lvalue`.

La solution est apportée par `std::forward`

similaire à `move`, mais `forward` est un cast conditionnel = cast en une `rvalue` si et seulement si son argument est initialisé avec une `rvalue`.

Reformulation : `forward` passe un objet à une autre fonction de façon à ce que l'argument conserve sa propriété originale de `lvalue` ou de `rvalue` (`move` convertit de manière non conditionnelle en `rvalue`).

Donc, ne pas utiliser `forward` à la place de `move`.

Note : `forward` nécessite d'indiquer le type du `forward`

exemple : `std::forward<std::string>(rhs.s)`

Les références universelles devraient conditionellement être castées en `rvalue`s (avec `forward`) lorsqu'elles sont transmises à d'autres fonctions, car elles sont parfois associées à des `lvalue`s.

Conseils :

- ne jamais utiliser `forward` sur la référence à une `rvalue` (code long, peut engendrer des erreurs), mais toujours sur une référence universelle.
- ne jamais utiliser `move` avec une référence universelle (peut modifier une `lvalue`, par exemple une variable locale, de manière inattendue).
- si un paramètre est passé comme référence universelle (resp. rvalue), il ne doit être transmis avec forward (resp. move) qu'une seule fois dans le code de la fonction = à sa dernière utilisation.
  Exemple :

```
Matrix operator+(Matrix&& lhs, const Matrix& rhs) {
    lhs += rhs;            // première utilisation
    return std::move(lhs); // dernière utilisation: move
}
```

10.4 Référence universelle et RVO

Si une fonction retourne une objet par valeur, et que l'objet retourné est associé à la référence d'une rvalue (resp. une référence universelle), alors appliquer `move` (resp. `forward`) lorsque la référence est retournée permet de transmettre la référence à l'emplacement où la fonction retourne sa valeur sous l'hypothèse que l'objet implémente le constructeur par déplacement (s'il ne l'implémente pas, une copie sera faite).

Exemple : avec la référence à une rvalue

```
Matrix operator+(Matrix&& lhs, const Matrix& rhs) {
    lhs += rhs;      return std::move(lhs);       }
```

lhs est déplacé à l'emplacement où la fonction retourne sa valeur (sans le move, lhs serait copié).

Exemple : avec une référence universelle

```
template<typename T> Fraction reduceAndCopy(T&& frac) {
    frac.reduce();
    return std::forward<T>(frac); // move si rvalue
}
```

Attention : ne jamais move/forward une variable locale d'une fonction au retour de cette fonction car la variable locale n'est alors plus candidate pour le RVO (i.e. NRVO non fonctionnel avec move/forward).

10.5 Remarque

Remarque : éviter si possible de surcharger une référence universelle

- une référence universelle est extrêmement gloutonne.
- elle capte tous les types qui ne sont pas exactement ceux de la surcharge.

Exemple :

```
class A {  private:    std::string name;
public:
   template<typename T> explicit A(T&& n)
                        : name(std::forward<T>(n)) {};
   A(const A& rhs) : default; // ctor par copie
   A(A&& rhs) : default;      // ctor par déplacement
};
```

- `A cp("Toto"); auto cloneOfP(cp)`
  erreur de compilation car `cp` n'est pas `const`, donc déclenche l'instanciation du constructeur par forwarding `A<T>(T&&)` avec `T=A`, puis tente de construire name (un `std::string`) à partir de la référence à un `A`, ce qui échoue.
- `const A cp("Toto"); auto cloneOfP(cp)`
  lance le constructeur par copie (car une fonction normale est toujours préférée à une fonction templatisée si le compilateur a le choix).

10.6 Perfect-forwarding

La transmission parfaite (ou perfect-forwarding) est, pour une fonction générique `pft`, l'action de passer ses arguments à une autre fonction `fun` sans perdre aucune information sur la catégorie ou la qualification de ses arguments.

A savoir,

- la fonction `template <tParams> rType pft(pftParams)` fait appel en interne à la fonction `fun(pftParams)`.
- le perfect-forwarding consiste à faire en sorte que l'appel `pft(args)` invoque en interne de manière exactement identique `fun(args)`.

Rappel : le paramètre d'une fonction est toujours une `lvalue`, même si son type est la référence à une `rvalue`. A savoir pour void `f(Object&& w)`, alors `w` est une `lvalue`, même si son type une `rvalue` référence à `Object`.

Ceci est donc réalisé par l'utilisation de :

- une fonction générique prenant en argument une référence universelle,
- la fonction `forward` qui permet d'effectuer un `move` sur une `rvalue`, et un passage par référence sur une `lvalue`.

11 Template et constexpr

Exemple : expression constante utilisant un template

```
// classe avec des constructeurs constexpr
class conststr {
    const char* p;
    std::size_t sz;
    public:
    template<std::size_t N>
    constexpr conststr(const char(&a)[N]): p(a), sz(N - 1) {}
    constexpr char operator[](std::size_t n) const {
        return p[n]; }
    constexpr std::size_t size() const { return sz; }
};
// comptage récursif du nombre de minuscules d'un conststr
constexpr std::size_t countlower(conststr s,
    std::size_t n = 0, std::size_t c = 0) {
    return (n == s.size()? c : (s[n] >= 'a' && s[n] <= 'z' ?
       countlower(s, n + 1, c + 1): countlower(s, n + 1, c)));
}
template<int n> struct constN {
    constN() { std::cout << n << '\n'; } };
```

```
// "Hello, world!" implicitement converti en conststr
constN<countlower("Hello, world!")> out;
```

12 Conclusion

Résumons les notions abordés dans ce chapitre :

- Les templates sont un outil essentiel pour limiter la réécriture de code, en écrivant du code générique,
- On peut définir des fonctions génériques, des méthodes génériques, et des classes génériques,
- Les types, les paramètres entiers, ... peuvent être rendu génériques et instanciés à une valeur connue à la compilation,
- Le code est généré par le compilateur et optimisé pour le type instancié,
- Il peut être spécialisé afin de l'adapté à certains types spécialisés,
- Il n'est pas possible de mélanger virtualité et généricité,
- Les traits de type peuvent être utilisés sur les types instanciés afin de vérifier qu'il vérifie certaines propriétés,
- Les functors sont un outils couramment utilisé afin de créer des objets fonctionnels avec des états internes.