INFO0402 : Méthodes de programmation orientée objet

Nouveauté du C++11

Pascal Mignot

2015-2016

UNIVERSITÉ
DE REIMS
CHAMPAGNE-ARDENNE

Introduction

Cette leçon est destinée à introduire certains nouveaux outils particuliers du C^{++}_{11} que nous n'avons pas encore pu introduire dans les contextes recontrés :

- La possibilité de faire **référence à une rvalue**, ce qui ouvre tout un champs à l'optimization des codes lors du passage de paramètres.
- Les **pointeurs intelligents** qui permet d'ajouter des outils haut-niveaux pour gérer les pointeurs (basé sur les templates),
- Les **lambda-expressions** qui permettent de construire des fonctionnelles au vol.

Terminologie des valeurs

qualification cv (cv-qualified)

- un type qui utilise le qualificateur `const` et/ou `volatile` est qualifié cv.
- **Exemples :** `const int`, `volatile bool` sont qualifiés cv.

objet nommé (named/unnamed)

- un objet ou une fonction dont le nom est donné par un identificateur est dit nommé.
- un objet ou une fonction auquel on ne peut pas faire référence par un nom est dit anonyme (ou non nommé).
- **Exemple :**

```
Stack    p;                  // p nommé
p = Stack(10);   // Stack(10) non nommé
```

Introduction au déplacement

Soit deux objets `src` et `dst`. On souhaite propager la valeur de l'objet `src` à l'objet `dst`.

Il y a deux façons d'effectuer cette propagation :

- **par copie :** propage la valeur de `src` vers `dst` sans modifier `src`.
- **par déplacement :** propage la valeur de `src` vers `dst` en modifiant possiblement `src`.

Exemple :

Considérons la classe suivante :

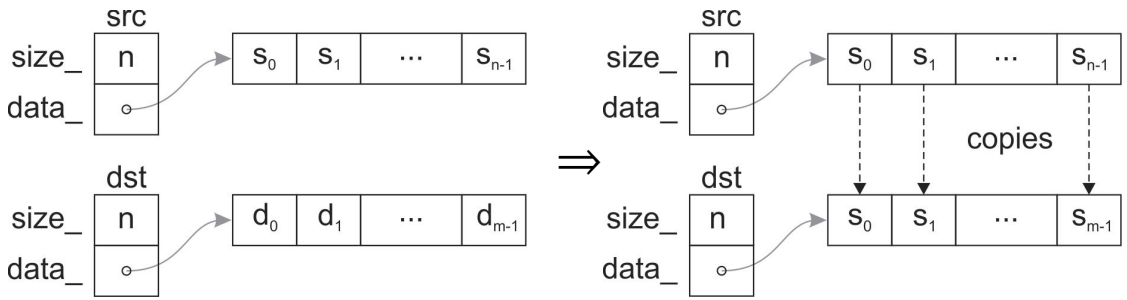
```
class Vector {
private:
  int    size_;  // taille du vecteur
  double *data_; // pointeur vers les données du vecteur
public: ...
};
```

Comment dans ce cas s'implémente la copie et le déplacement ?

Introduction au déplacement

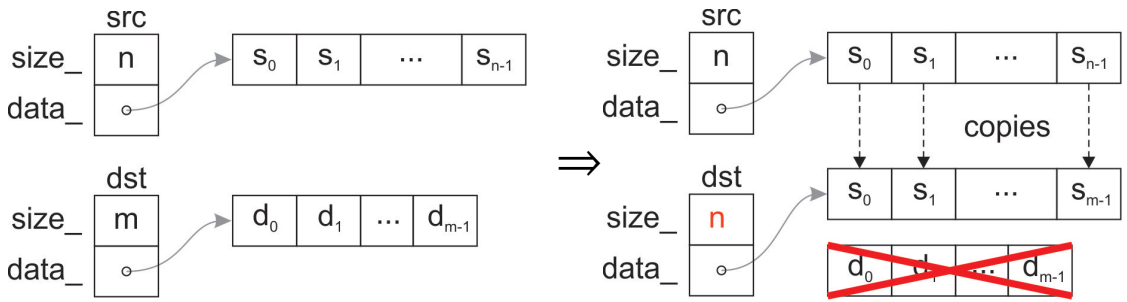
Exemple : (suite)

- cas 1 : copie de vecteurs de même taille.



La copie seule des données suffit.

- cas 2 : copie de vecteurs de tailles différentes.



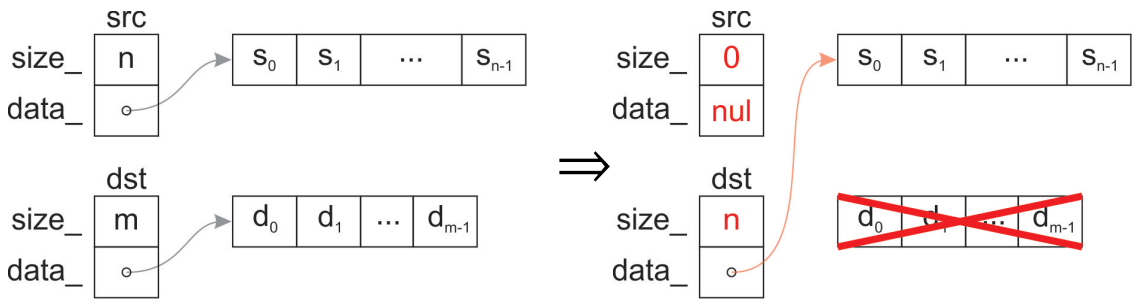
Dans ce cas, il faut pour `dst` :

- déallouer le tableau des données,
- allouer un nouveau tableau de taille `src.size_`,
- mettre à `dst.size_` avec `src.size_`

Introduction au déplacement

Exemple : cas 3 : déplacement (méthode 1)

méthode 1 = destruction et déplacement



Il faut donc :

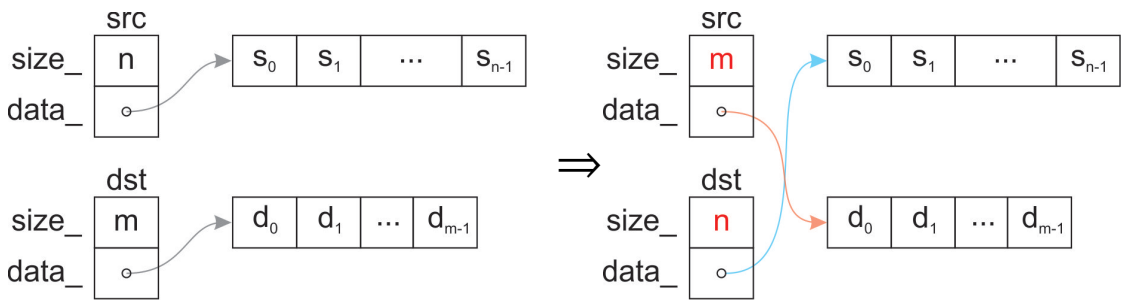
- déallouer le tableau des données de `dst`,
- copier `src` dans `dst`
- mettre `src` à 0 (*i.e.* `(size_,data_) = (0,nullptr)`)

Dans ce cas, le vecteur `src` est vide après le déplacement.

Introduction au déplacement

Exemple : cas 3 : déplacement (méthode 2)

méthode 2 = permutation



Il faut permuter `src` et `dst`, c'est-à-dire :

- permuter `src.size_` et `dst.size_`
- permuter `src.data_` et `dst.data_`

Dans ce cas, le vecteur `src` contient le vecteur `dst` (et inversement).

L'avantage de cette méthode, et que, si on manipule en général des vecteurs de même taille, la place mémoire peut être facilement recyclée.

La désallocation des anciennes données de `dst` est donc reportée à la libération de `src`.

Introduction au déplacement

Conséquences :

- En général, un déplacement est plus efficace qu'une copie (souvent de beaucoup, dès qu'une partie des données est stockée à l'extérieur de l'objet).
- Si le code qui suit le déplacement ne dépend pas de la valeur de l'objet source, une copie peut être remplacée de manière sûre par un déplacement.

Comment fournir un moyen au langage tel qu'un déplacement soit effectué automatiquement chaque fois que le contexte garantit que ce déplacement sera sûr ?

Il serait également souhaitable de pouvoir ignorer le comportement par défaut et de forcer un déplacement dans les cas où la sureté du déplacement est garantie par le programmeur (du fait de sa connaissance du comportement du programme) et non par le contexte.

Les références sur les `rvalues` sont le moyen de fournir ce mécanisme.

Typologie des valeurs

La sémantique de déplacement fournit un moyen de déplacer le contenu d'un objet entre objet, plutôt que de le copier.

A savoir, si la sémantique de déplacement est définie sur un objet, alors :

- le retour d'un objet par valeur déplace la valeur renvoyée dans l'objet accueillant le résultat (plutôt que de le copier).
- à l'appel d'une méthode, le passage en paramètre d'un objet temporaire peut le déplacer.
- l'appel explicite à la sémantique move permet de déplacer de déplacer un objet.

Exemple :

```
// construction par déplacement de l'objet retourné
Object v = MakeObject(1);
ObjectList L;
// déplacement de l'objet temporaire dans la liste
L.addObject( MakeObject(3) );
// déplacement de l'objet construit dans la liste
L.addObject( std::move(v) );
// ici, v est vide
```

Typologie des valeurs

Toute expression C++ peut être caractérisée par deux propriétés indépendantes : son type et la catégorie de sa valeur.

Les valeurs sont catégorisées suivant leurs propriétés de :

- **identifiabilité** : s'il est possible de déterminer si une expression fait référence à la même entité d'une autre expression (tel qu'en comparant les adresses des objets ou des fonctions qu'elles identifient, de manière directe ou indirecte).
- **déplaçabilité** : s'il est possible de déplacer la valeur dans le contexte (*i.e.* si le [constructeur par déplacement | assignation par déplacement | surcharge de fonction qui implémente la sémantique de déplacement] peut être attaché à l'expression).

On définit alors deux catégories primaires :

- une `glvalue` est une expression `identifiable` (=généralized lvalue ; auparavant nommée `lvalue`).
- une `rvalue` est une expression **déplaçable**.

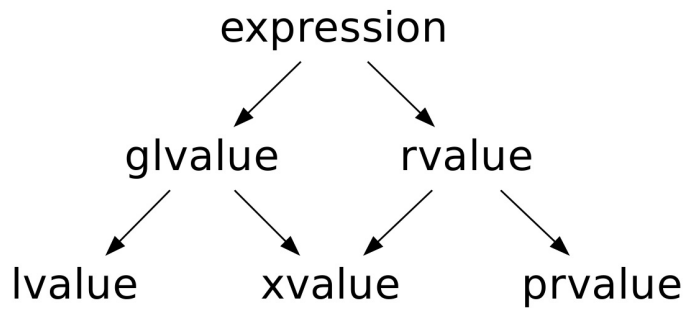
Typologie des valeurs

Ces deux catégories primaires peuvent être à leur tour subdivisées en trois catégories :

- `lvalue` = expression identifiable mais non déplaçable.
- `xvalue` = expression identifiable et déplaçable
- `prvalue` = expression non identifiable mais déplaçable

Note : dernière catégorie possible non utilisée (ni identifiable et ni déplaçable).

Ces catégories se hiérarchisent donc de la manière suivante :



Note : la nouvelle définition de la `lvalue` ne se superpose pas avec l'ancienne (voir exemple plus loin).

Rappels : BIT = build-in type = `int`, `float`, `bool`, `char`, ...

Typologie des valeurs

Exemples :

```
int a = 5;          // a=I/ND=lvalue, 5=NI/D=prvalue
int &b = a;         // b=I/ND=lvalue, a=I/ND=lvalue
int c = (a+4)/2;    // c=I/ND=lvalue, (a+4)/2=NI/D=prvalue
// int &d = 5;      // erreur: 5 NI (non adressable)
// int &e = a/2;    // erreur: a/2 NI
```

Rappel du sens de &b :

- &b fait référence à un entier (= est un autre nom pour).
- adresse de b = adresse de a.
- une référence est en fait une référence à une `lvalue`.

Noter que :

```
const int &d = 5;       // valide
const int &e = (a+4)/2; // valide
```

car `const int&` signifie référence vers un entier constant (et non référence constante vers un entier).

Donc, une référence à une constante est le seul outil dont nous disposons qui permet de faire référence à un objet temporaire. Malheureusement, elle n'est pas déplaçable car constante.

Nous avons donc besoin d'un autre outil.

Typologie des valeurs

Nouveau concept C^{++}_{11} : référence à une `rvalue`.

Nouveau modificateur de type : `&&`

`T&&` = référence à une `rvalue` de type `T`.

Une référence à une `rvalue` fait référence au résultat d'une expression temporaire dont l'assignation ne persiste pas au-delà de l'expression qui la définit (une variable de ce type est une lvalue).

C'est un autre nom pour la place temporaire qu'occupera le résultat résultant de l'évaluation de l'expression.

Exemples :

```
int   funPR(), &funL(), &&funX(); // déclaration fonction
int a = funPR();   // funPR()=NI/D=prvalue,  a=I/ND=lvalue
int &b = funL();   // funL()=I/ND=lvalue,    b=I/ND=lvalue
int &&c = funX();  // funX()=I/D=xvalue,     c=I/ND=lvalue
```

Une variable de type `T&&` a donc pour vocation d'être la référence à une variable de type `T` qui peut être déplacée.

Typologie des valeurs

Propriétés des `glvalue` :

- peut être implicitement convertie en une `prvalue` (conversion implicite `lvalue`→`rvalue`, `tableau`→`pointeur`, `fonction`→`pointeur`).
- peut être polymorphique (le type dynamique de l'objet n'est pas nécessairement le type de l'expression),
- peut avoir un type incomplet.

Propriétés des `rvalue` :

- l'adresse d'une `rvalue` ne peut pas être prise (pas identifiable),
- ne peut pas être utilisé comme opérande de gauche dans une assignation (même raison),
- peut être utilisée pour initialiser une référence à une `rvalue` ou à une référence constante à une `lvalue` (la durée de vie de la rvalue utilisée pour l'initialisation est alors prolongée jusqu'à la fin de portée de la référence).
- (pour plus tard) si une fonction `F` a deux surcharges `F(T&&)` et `F(const T&)`, alors l'appel `F(rvalue)` appelle `F(T&&)`.

Typologie des valeurs

`lvalue`

- Ce nom a été gardé pour des raisons historiques mais le sens ne recoupe pas l'ancien sens (à savoir, expression à gauche d'un =).
- **Rappel définition :** expression identifiable et non déplaçable.
- **Comment reconnaître une `lvalue` ?** si elle peut être utilisée (ou est utilisée) pour stocker une valeur. Donc, une expression est une `lvalue` si :
  - il est possible de prendre l'adresse de l'expression,
  - c'est une référence sur une `lvalue`.
- **Exemples de `lvalue` :**
  - tout nom de variable ou de fonction dans la portée est une lvalue (y compris si son type est référence à une `rvalue`, cf plus tard).
  - les chaines de caractères littérales (exemple : `"Toto"`),
  - si `p` est un pointeur, `*p` est une `lvalue`, `p[n]` est une `lvalue`,
  - `a.m` ou `p->m` où `m` est un membre mais pas un membre énuméré (exemple : `v` dans `struct S { enum{ v = 0 } };`) ou une méthode.
  - les assignations (exemple : `a=b` dans `a=b=c`),
  - un appel de fonction (surcharge d'opérateur, cast, incrémentation préfixes de BITS inclus) qui retourne un type qui est une référence à une lvalue (exemple : `static_cast<int&>(x)`).
  - un appel de fonction (surcharge d'opérateur, cast inclus) qui retourne un type fonctionnel dont le type de retour est une référence sur une rvalue (exemple : `static_cast<void (&&)(int)>(x)`).
  - un cast vers un type qui est une référence à une lvalue ou une rvalue.

Typologie des valeurs

Exemples : `lvalue`

```
// définitions
int fun();
const int i = 1;
int j = 0;
char buffer[] = "Hello"; // "Hello" lvalue
char * s = buffer;
char &Car(char *v, int i) { return v[i]; }
int &&k = j+3;
```

```
// objets et fonctions nommés
j = i+1;          // i,j lvalue
fun();            // fun lvalue (pas fun())
// pointeur déréférencé
*s = 'a';         // *s lvalue
*(s+1) = 'b';     // *(s+1) lvalue
// retour d'une reférence
j=Car(s,3);       // Car(s,3) lvalue
// référence à rvalue nommée
j=k;              // k lvalue
```

Donc **attention**, avec cette définition, une `lvalue` peut apparaître à droite d'un =.

Typologie des valeurs

`prvalue` :

- `prvalue` pour "pure rvalue", fait référence à un objet, en général proche de sa fin de vie (tel que sa ressource peut être déplacé) et est le résultat de certains types d'expression impliquant la référence à une `rvalue`.
- **Rappel définition :** expression non identifiable mais déplaçable.
- **Exemples de `prvalue` :**
  - tout littéral (32, `false`, `nullptr`, ...) sauf les chaînes de caractères,
  - un appel de fonction (surcharge d'opérateur inclus) qui retourne un type qui n'est pas une référence (exemple : `sin(x)`).
  - toute expression arithmétique ou logique utilisant des types et opérateurs définis par défaut (exemple : `a+b` ou `u!=4` sur des BITs).
  - l'incrémentation ou la décrémentation postfixe,
  - `a.m` ou `p->m` où `m` est un membre statique énuméré ou une méthode non statique.
  - `this`,
  - un cast vers un type qui n'est pas une référence,
  - une lambda-expression
- le type d'une `prvalue` est toujours celui de l'expression (donc n'utilise pas le polymorphisme, et son type ne peut pas être incomplet).
- ne peut pas être qualifié cv.

Typologie des valeurs

Exemples : prvalue

```
// définitions
Stack    s;
int incr(int k) { return k++};
```

```
// objets temporaires
s = Stack(10);    // Stack(10) prvalue
// littéraux
int i = 42;       // 42 prvalue
// fonction ne retournant pas de référence
i = incr(i);      // incr(i) prvalue
// résultat d'expression arithmétique
i = 4*i+8;        // 4*i+8 prvalue
// résultat d'expression logique
bool v = (i==5); // (i==5) prvalue
// incrémentation postfixe
// rappel: renvoie la valeur incrémentée
i++;              // i++ prvalue
```

Typologie des valeurs

`xvalue` :

- `xvalue` pour "expiring value", fait référence à un objet, en général proche de sa fin de vie (tel que sa ressource peut être déplacé) et est le résultat de certains types d'expression impliquant la référence à une `rvalue`.
- **Rappel définition :** expression identifiable et déplaçable.
- **Exemples de `xvalue` :**
  - un appel de fonction (surcharge d'opérateur inclus) qui retourne une référence à une `rvalue` (exemple : `T&& move(T&&)`).
  - un cast vers un type qui est une référence vers une `rvalue`.
  - `a[n]` où `a` est un tableau `rvalue`,
  - `a.m` où `a` est un objet `rvalue`, et `m` un membre non statique qui n'est pas une référence,
- une `xvalue` correspond donc :
  - soit à une `lvalue` convertie explicitement en une référence à une `rvalue` avec une fonction appropriée (typiquement `std::move`), façon d'indiquer explicitement qu'elle est en fin de vie.
  - soit à un objet qui fait référence à une `rvalue`
- une `xvalue` peut être polymorphique et être qualifié cv.

Déplacement et catégorie

Avec ces catégories, un déplacement sur :

- une `lvalue` n'a pas la garantie d'être sûre, puisque le code peut accéder à l'objet associé à travers son nom, un pointeur ou une référence.

  donc, une `lvalue` ne doit jamais être automatiquement déplacée.
- une `prvalue` a toujours la garantie d'être sûre (puisqu'une prvalue n'a pas d'identité).
- une `xvalue` est considérée comme être sûre car elle est soit déjà une référence à une `rvalue`, soit a été explicitement convertie en une `xvalue` (et donc l'utilisateur considère que le déplacement est sûr).

Donc,

- si la source est une `rvalue` (une `prvalue` ou une `xvalue`), utiliser un déplacement à la place d'une copie est sûre,
- si la source est une `lvalue`, utiliser un déplacement à la place d'une copie n'a pas la garantie d'être sûre.

On veut donc que le langage utilise automatiquement le déplacement pour une `rvalue`, et la copie pour une `lvalue`.

Sémantique de déplacement

Sémantique de déplacement :

- une variable de type `T&&` (qualifiée cv ou non) est utilisée pour stocker une `rvalue` jusqu'à la fin de la portée de la variable,
- un paramètre de type `T&&` (qualifiée cv ou non) est utilisée pour indiquer que le paramètre est une `rvalue` et peut être déplacé au cours de l'appel de la fonction.

Le déplacement est typiquement implémenté de trois manières différentes :

- dans une classe T,
  - avec le constructeur par déplacement (typiquement `T(T&&)`) : permet de construire un objet en déplaçant une `rvalue`,
  - avec l'assignation par déplacement (typiquement `T& operator=(T&&)`) : permet d'assigner un objet en déplaçant une `rvalue`.
- explicitement, avec `T&& std::move(T&&)` (cette fonction indique seulement que l'objet passé en argument peut être déplacé, mais n'effectue pas le déplacement).
- mais également, dans toute fonction qui souhaite implémenter le déplacement de l'un de ses paramètres.

Élision de copie

Rappel [Return value optimization (RVO)] : Technique d'optimisation du compilateur qui élimine la copie de l'objet local retournée **par valeur** par une fonction vers l'objet de la fonction qui l'a appelée.

A savoir, le code suivant devrait :

```
A fun() { ... return A(k); }
void call() { ... A y=fun(); }
```

sans RVO :

- à la fin de la fonction `fun`, un objet temporaire est construit et retourné,
- la valeur du nouvel `y` est construit par copie à partir de l'objet temporaire, puis l'objet temporaire est détruit.

avec RVO :

- l'objet retourné doit être un temporaire fourni en argument du `return` (=construction de l'objet dans le return).
- le retour de la fonction est construit directement dans `y`.
- gain : une construction par copie + une destruction.

Élision de copie

Rappel [Named Return value optimization (NRVO)] : Même idée que le RVO, mais améliorée de manière à ce que l'on puisse retourner le nom d'un objet local.

A savoir, le code suivant devrait :

```
A fun() { ... A a; ... return a; }
void call() { ... A y=fun(); }
```

sans NRVO : idem RVO/sans RVO.

avec NRVO :

- un objet local nommé est construit, éventuellement modifié, puis retourné par la fonction.
- l'objet local est construit directement à l'emplacement de l'endroit où il est retourné (*i.e.* y dans le cas ci-dessus).
- limitation : l'objet local ne doit pas être un paramètre ou volatile, paramètre d'un catch, et être de même type que le type de retour.

Remarque : si l'élision de copie (=RVO ou NRVO) n'est pas possible, le `return` essaye une construction par déplacement, et si elle n'est pas définie, une construction par copie.

Élision de copie

Attention : l'élision de copie est la seule forme d'optimisation autorisée par le standard qui peut avoir des effets de bord visibles :

- Certains compilateurs n'effectuent pas l'élision dans tous les cas où elle est possible et/ou autorisée (exemple : compilation en mode debug),
- Existence de cas où elle n'est pas effectuée (cf NRVO) + argument d'un throw/catch dépend de la version du C++ (oui si C++ $\geq$C++$_{11}$, non avant).
- Ne pas se baser sur cet effet de bord, et construire attentivement le constructeur par copie/déplacement (sinon code non portable).

  A savoir, faire en sorte que construction + copie = construction directe
- L'ignorance de cette règle relève de l'erreur de conception.

Exemple :

```
struct Foo {
public: int _a{}; // défaut=0
    Foo(int a) : _a{a} {}
    Foo(const Foo &) {}
};
```

```
Foo a{10};
Foo bar = 10; // équivalent = Foo{10}
// b._a = 0  // sans élision de copie
// b._a = 10 // avec élision de copie
```

Déplacement et catégorie

Exemple : avec sémantique copie + déplacement

```
struct A { ...
 A();                 // Cd
 A(int x);            // Ci
 A(const A& x);       // Cc
 A(A&& x);            // Cm
 ~A();                // D
 A& operator=(const A&); // O=c
 A& operator=(A&&);      // O=m
};
A operator+(const A&, const A&); // O+
A fun(int);                      // avec élision [E]
A fun(char);                     // sans élision
```

```
A a1;          // Cd
A a2(2);       // Ci
A a3(a1);      // Cc
A a4 = a1;     // Cc
A a5(A(1));    // Ci
```

```
a1 = a2;     // O=c
a1 = A(2);   // Ci|O=m|D
a1 = 1;      // Ci|O=m|D
a1 = a2 + a3;
     // O+=>Ci[E]|O=m|D
```

```
A       f2();      // A f2(void)
A       f1(A());   // A f1(A(*)(void))
A   a1(std::move(A())); // Cd|Cm|D
A   a2(a2 + a3);        // 0+=>Ci[E]
```

```
A a1 = fun(1);   // Ci[E]
A a2 = fun('a'); // Cd|Cm|D
a1 = fun(2);     // Ci|O=m|D
a2 = fun('b');   // Cd|Cm|D|Om=|D
```

Constructeur par déplacement implicite

CPD implicitement déclaré :

- **CPD trivial :** constructeur inline qui effectue la même action que le CPC trivial (= copie mémoire de l'objet).
- **CPD implicitement déclaré :** lorsque l'utilisateur ne fournit ni constructeur par copie ou déplacement, ni assignation par copie ou déplacement, ni destructeur,
- **CPD implicitement effacé :** le CPD implicitement déclaré est effacé si :
  - le constructeur par déplacement est explicitement effacé (=delete),
  - la classe `T` a des membres virtuels ou des classes de bases virtuelles,
  - le constructeur par déplacement de toute classe dont `T` hérite, ou tout membre non statique de `T` n'est pas trivial.
- **CPD implicitement défini :** s'il est implicitement déclaré, mais pas implicitement effacé, et a pour code le CPC trivial inline.

Remarques :

- attention aux élisions de copie pouvant être utilisée pour l'optimisation du constructeur par déplacement.
- dans le CPD d'un objet composé, attention à effectuer un `std::move` dans les arguments dans la liste d'initialisation afin de lancer le CPD du sous-objet (voir la section sur le perfect forwarding).

Constructeur par déplacement et exceptions

Si une exception est lancée, le CPD ne sera pas en mesure de libérer la mémoire associé à l'objet temporaire passé en paramètre (puisqu'il est sensé être déplacé après l'appel à ce constructeur).

Conséquence : Un CPD ne doit pas lancer d'exception.

Pour l'assurer :

1. utiliser le mot-clef `noexcept` à la déclaration pour indiquer que le constructeur ne lance pas d'exception (i.e. `T(T&&) noexcept`),
2. utiliser `std::move_if_noexcept` à la place de `std::move`

   `move_if_noexcept<T>(x)` = `static_cast<T&&>(x)` si `T(T&&)` est `noexcept` et `static_cast<T&>(x)` sinon.

   Donc, le CPD est lancé s'il ne lance pas d'exception, et le CPC sinon.

Exemple :

```
struct T { ...
    T(T&& t) noexcept { ... }
    T(T& t) noexcept { ... }
};
```

```
struct U { ...
    U(U&& u) { ... }
    U(U& u) { ... }
};
```

```
T a, b = std::move_if_noexcept(a);      // lance le CPD
U c, d = std::move_if_noexcept(c);      // lance le CPC
```

Constructeur par déplacement

Principe : pour un type `T`, un constructeur par déplacement a pour prototype `T(T&&)`.

Exemple :

```
// on reprend l'exemple de la classe vector
class Vector {
private: size_t size_;  float *data_;
public: ...
    // constructeur par déplacement
    T( T &&v) :
        size_(v.size_),
        data_(v.data_) {
        v.size_ = 0;
        v.data_ = nullptr;
    }
    // assignation par déplacement
    T& operator=(T &&v) {
        std::swap<size_t>(size_,v.size_);
        std::swap<float*>(data_,v.data_);
        return *this;
    }
};
```

Sémantique de déplacement

Comme nous l'avons déjà vu, si un paramètre d'une fonction est une référence sur une rvalue, alors il peut être déplacé (donc modifié).

Si le contexte permet de déterminer qu'il est possible de déplacer une `lvalue` lors de l'appel d'une fonction, alors il faut que :

- cette `lvalue` soit explicitement convertie en référence sur une `rvalue` lors du passage de l'argument : effectué avec la fonction `std::move`.
- la `rvalue` obtenue soit passée à une fonction dont une surcharge gère le déplacement (= accepte comme paramètre une référence à une rvalue).

Note : la fonction `std::move` ne fait rien d'autre que de convertir l'argument en une référence à une rvalue, le rendant ainsi candidat pour un déplacement, mais n'effectue pas le déplacement.

Exemple :

```
int fun(A &&x) { ... }
int fun(const A &x) { ... }
```

```
A   x;
int y1 = fun(x);            // call fun(A&)
int y2 = fun(std::move(x)); // call fun(A&&)
```

Sémantique de déplacement

Application : template swap

Typiquement, l'implémentation typique de swap est la suivante :

```
template <class T> void swap(T &x, T &y) {
                        T tmp(x); x=y; y=temp; }
```

Noter que ce code nécessite :

- un constructeur par copie, et deux copies par assignation, donc particulièrement inefficace pour beaucoup de type `T`.
- le type `T` doit être copiable (i.e. CPC et APC existent).

Avec C_{11}^{++}, le code de swap est désormais :

```
template <class T> void swap(T &x, T &y) {
   T tmp(std::move(x)); // CPD si possible, CPC sinon
   x=std::move(y);      // APD si possible, APC sinon
   y=std::move(temp);   // APD si possible, APC sinon
}
```

Ce code convertit tous les arguments des constructeurs ou assignations en référence sur une `rvalue` avec `std::move` de façon à permettre aux fonctions appelées d'effectuer un déplacement s'il est implémenté, et ainsi éviter les copies.

Sémantique de déplacement

Remarque : attention au retour par valeur

- **éviter de le retour par valeur constante :**

  Exemple : `const std::string getMsg() { return "Hello"; }`

  L'objet constant retourné ne peut pas être utilisé comme une source pour un déplacement (`const` donc non modifiable).

  ⇒ Mauvaise interaction avec la sémantique de déplacement.

- **ne jamais mettre un `std::move` lors d'un retour par valeur.**
  Exemple : `A getA() { return std::move(A()); }`

  Or, le type retourné par `move` est `A&&` et non `A` (donc différent).

  En conséquence, la RVO/NRVO ne peut pas être appliquée.

Conséquences :

- un paramètre qui doit être déplacé ne doit pas être déclaré comme constant, sinon il sera copié au lieu d'être déplacé,
- `std::move` ne déplace rien, et ne garantit pas que l'objet sera déplacé, ni qu'il sera éligible pour un déplacement.
- une fonction qui prend en paramètre une référence sur une `rvalue` ne garantit pas que l'objet passé sera déplacé.

Problèmes

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 `lvalues` ou des `rvalues`, il faudrait écrire 2^n fonctions pour gérer toutes les combinaisons de `lvalues` et de `rvalues`.

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

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&&
```

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)`

Référence universelle

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

Référence 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 `lvalues`.

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`.

Référence universelle

`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 rvalues (avec `forward`) lorsqu'elles sont transmises à d'autres fonctions, car elles sont parfois associées à des lvalues.

Référence universelle

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
}
```

Référence universelle

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).

Référence universelle

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).

Sémantique de déplacement

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`.

Passage de paramètres

Surcharges possibles pour un paramètre :

Catég.	Type	lvalue	const lvalue	rvalue	const rvalue
value	`T`	x	x	x	x
	`const T`	x	x	x	x
lvalue	`T&`	x			
	`const T&`	x	x	x	x
rvalue	`T&&`			x	
	`const T&&`			x	x

En plus des règles suivantes :

1. au sein d'une même catégorie, une surcharge est conflictuelle,
2. un paramètre de catégorie value ne peut être surchargé par un paramètre de catégorie `lvalue` ou `rvalue`,
3. un paramètre de catégorie `lvalue` peut être surchargé par un paramètre de catégorie `rvalue`.

Rappel : un `const T&` peut accueillir une `rvalue` sous la forme de la référence à l'espace mémoire temporaire qui stocke cette `rvalue` (et éventuellement prolonge sa durée de vie).

Durée de vie des objets temporaires

Comment lire ce tableau ?

- `fun(const T)` ou `fun(const T&)` acceptent tout paramètre de type `T`.
- `fun(T)` et `fun(const T)` ambigües.
- `fun(const T&)` et `fun(T&&)` non ambigües (règle 3) : `fun(const T&)` = `lvalues` et `fun(T&&)` = `rvalues`.
- `fun(T&)` et `fun(T&&)` non ambigües (règle 3) : `fun(T&)` = `lvalues` non constantes et `fun(T&&)` = `rvalues`.

Cas des pointeurs :

déclaré \ passé	`T*`	`const T*`
`T*`	x	
`const T*`	x	x

L'ajout de `const` derrière le type déclaré ou passé (= pointeur constant) ne change rien.

i.e. un `T* const` peut être passé à la place d'une `T*`, et vice-versa.

Qualification de référence

Dans une classe, la qualification de référence d'une méthode est un moyen d'indiquer :

- avec `const`, que la méthode laisse l'objet sur lequel elle s'exécute constant,
- avec & (resp. &&), que la méthode (non statique) s'exécute sur un objet sous forme d'une `lvalue` (resp. `rvalue`).

Exemple :

```
class A { ...
public:
    void View() const;
    void doWork() &;    // si *this lvalue
    void doWork() &&;   // si *this rvalue
};
A makeA();
```

```
A     a;
a.doWork();          // appel doWork()&
makeA().doWork();    // appel doWork()&&
```

Qualification de référence

Remarques :

- ces qualifications permettent notamment d'effectuer des moves dans le cas des `rvalues`,
- la surcharge de deux fonctions membres avec les mêmes types de paramètres requiert que les deux aient des qualificateurs de référence ou qu'elles en soient dépourvues,
- une qualification `const &` ou `const &&` est possible.
- un destructeur ne doit pas être déclaré avec un qualificateur de référence.

Exemple :

```
class Y {
    void h() &;
    void h() const &; // OK
    void h() && ;     // OK
    void i() &;
    void i() const;   // erreur: pas de qualif. ref.
};
```

Pointeur intelligent

Il existe trois types de pointeur intelligent en C^{++}_{11} :

- `unique_ptr` : personnification de la sémantique de propriété exclusive,
- `shared_pointer` : personnification d'un objet auquel plusieurs pointeurs font références, et dont la durée de vie dépend du nombre de pointeurs qui y font référence.
- `weak_ptr` : pointeur intelligent qui a une référence non propriétaire sur un objet partagé par un `shared_pointer`.

L'utilisation de pointeur intelligent est le plus souvent d'assurer à la fois :

- une libération automatique des ressources associées au pointeur en fin de portée,
- une gestion de la mémoire sans fuite mémoire, même en cas d'exception,
- une gestion du multithreading à condition que les constructeurs appropriés soient utilisés.

Donc, ne pas hésiter à les utiliser.

unique_ptr

personnification de la sémantique de propriété exclusive :

- possède la ressource sur laquelle il pointe,
- **déplacement :** déplace la propriété du pointeur source vers le pointeur de destination (la source est ensuite mise à null, donc ne doit pas être const),
- **copie :** non autorisée (sinon deux pointeurs vers la même ressource).
- **destruction :** détruit la ressource si le pointeur est détruit ou s'il est assigné à une autre ressource.
- un `const unique_ptr` a une vie limitée à la portée dans laquelle il est déclarée.

Exemple :

```
int main() {
    std::unique_ptr<A> p1(new A);   // p1 possède A
    if (p1) p1->bar();
    {   // propriété transférée à p2
        std::unique_ptr<A> p2(std::move(p1));
        f(*p2);
        // propriété retournée tout p1
        p1 = std::move(p2);
    }
    if (p1) p1->bar();
}   // fin de portée p1: destruction
```

unique_ptr

Caractéristiques :

- la taille d'un `unique_ptr` est celle d'un pointeur standard.
- **modificateurs :**
  - `release` : retourne l'objet géré et relâche la propriété,
  - `reset` ou `operator=` : remplace l'objet géré
  - `swap` : échange l'objet géré
  - utiliser `std::move` pour tranférer la propriété.
- **observateurs :**
  - `get` : retourne un pointeur vers l'objet géré
  - `operator bool` : vérifie si le pointeur est affecté
- **gestion d'exception :** garantie la suppression de l'objet sur sortie normale ou suite à une exception.

Notes :

- `std::make_unique` donne un template rendant la construction d'un `unique_ptr` plus facile (C^{++}_{14}, dispo. dans VS2015)

```
// unique_prt sur un A (avec constructeur par défaut)
std::unique_ptr<A> a1 = std::make_unique<A>();
// unique_prt sur un A (avec constructeur spécialisé)
std::unique_ptr<A> a2 = std::make_unique<A>(0, 1, 2);
```

- un destructeur spécifique peut être passé au constructeur, mais fait passer la taille `unique_ptr` d'un word à deux words.

unique_ptr

Cas des tableaux :

- utiliser le constructeur : `std::unique_ptr<T[]>` ou `std::make_unique<T[]>`,
- accès aux éléments : utiliser `operator[]`.

Exemple 1 :

```
std::unique_ptr<A[]> p1(new A[50]);
std::unique_ptr<A[]> p2 = std::make_unique<A[]>(50);
p1[0]=4;
```

Exemple 2 :

```
class A {
private : std::unique_ptr<int[]> x, y;
public :
  A(std::size_t xSize, std::size_t ySize):
        x(std::make_unique<int[]>(xSize)),
        y(std::make_unique<int[]>(ySize)) {}
  ~A() {}  // utilise le destructeur de unique_prt
  // ...
};
```

shared_ptr

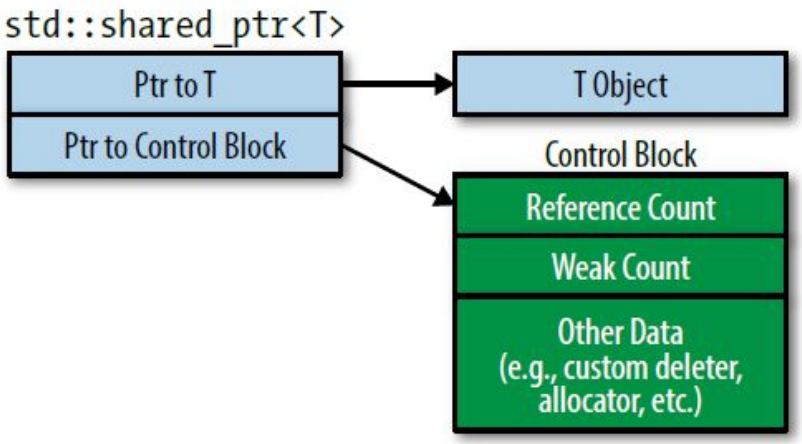
Personnification d'un objet auquel plusieurs pointeurs font références, et dont la durée de vie dépend du nombre de pointeurs qui y font référence.

- l'objet n'est possédé par aucun pointeur qui y fait référence,
- l'objet est automatiquement détruit lorsqu'il n'y a plus aucun pointeur qui pointe vers lui (destruction déterministe sans garbage collector) à travers un compteur de référence,
- un constructeur incrémente le compteur, un destructeur le décrémente. Note que l'incrémentation et la décrémentation des références doit être fait de manière atomique (=support multi-thread), ce qui est coûteux.
- **copie par assignation :** sp1=sp2 (fait pointer sp1 vers l'objet de sp2 - donc incrémente le compteur de l'objet référencé, mais s'il fait lui-même référence à un autre objet, le compteur de cet objet est décrémenté.
- **déplacement (move) :** sp1 -> sp2, ne change pas le compteur de référence de l'objet pointé par sp1, mais décrémente celui de sp2 s'il n'est pas nul. En conséquence, une move est plus rapide qu'une copie.
- un version spécialisée de `std::swap` existe et permet d'échanger deux pointeurs.

shared_ptr

Caractéristique :

- la taille d'un `shared_pointer` est le double d'un pointeur standard. L'utilisation d'un destructeur spécifique ne change par la taille d'un `shared_pointer` (voir la structure ci-dessous)
- le bloc de contrôle est créé et initialisé par le premier pointeur partagé créé sur l'objet (rappel : deux threads peuvent demander en même temps la création du pointeur partagé),
- même chose si un `shared_pointer` est créé à partir d'un `unique_ptr`.



shared_ptr

Attention :

- `std::shared_ptr<T>(new T(args))` effectue (au moins) deux allocations (une pour le bloc de contrôle et une pour l'objet), ce qui poser des problèmes en terme de gestion d'exception et/ou de thread-safety.
- toujours utiliser `std::make_shared` pour construire un `shared_ptr<T>` en une seule allocation (plus efficace, thread-safe, gestion d'exception).
- la gestion d'un objet `std::shared_ptr<T>` est garantie d'être thread-safe, mais pas l'objet possédé lui-même.

Exemple :

```
std::shared_ptr<int>  *a = std::make_shared<int>()

std::vector<std::shared_ptr <std::string>> all;
all.emplace_back(std::make_shared <std::string>("un"));
all.emplace_back(std::make_shared <std::string>("deux"));
all.emplace_back(std::make_shared <std::string>("trois"));
std::vector<std::shared_ptr <std::string>>
                    some(all.begin(), all.begin() + 2);
for(auto &x:all)
   std::cout << *x << "=1" << x.use_count() << std::endl;
// sortie: one=2 two=2 three=1
```

Remarque :

un `shared_pointer` a un surcoût par rapport à un `unique_ptr` : toujours

weak_ptr

Un `weak_ptr` est un pointeur intelligent qui a une référence non propriétaire (faible) sur un objet partagé par un `shared_pointer`.

Plus précisément,

- à la construction, le `weak_ptr` est initialisé avec un `shared_pointer` (peut lancer une exception `std::bad_weak_ptr` n'est pas valide).
- quand l'on souhaite accéder à un objet seulement s'il existe, et qu'il peut être effacé à n'importe quel moment par un autre thread, `weak_ptr` permet de suivre l'objet :
  - si l'objet existe, alors il est converti en `shared_pointer` pour assurer une propriété temporaire,
  - sinon, l'objet doit être considéré comme expiré.

Méthodes :

- `expired` permet de vérifier si l'objet a déjà expiré,
- `lock` acquiert un verrou sur l'objet référencé (retourne un `shared_pointer`
- `swap` pour permuter deux `weak_ptrs`.

weak_ptr

Utilisation :

1. déclarer un `weak_ptr` initialisé avec un `shared_pointer`,
2. l'appel à la méthode `lock()` sur le `weak_ptr` renvoie un `shared_pointer` propriétaire si l'objet existe encore, et un `shared_pointer` nul sinon.

   la propriété n'est conservée que jusqu'à la fin de la portée de l'objet `shared_pointer` acquis.

Exemple :

```
std::shared_ptr<int> sptr = std::make_shared<int>(10);
std::weak_ptr<int> weak1 = sptr;
sptr.reset( new int(5) );
std::weak_ptr<int> weak2 = sptr;
// ici: auto = std::shared_ptr<int>
if (auto tmp1 = weak1.lock()) { /* valide */ }
else { /* expiré */ }
if (auto tmp2 = weak2.lock()) { /* valide */ }
else { /* expiré */ }
```

`tmp1`/`tmp2` expirent à la fin de la conditionnelle.

Idiome **pImpl**

Idiome qu'il est important de connaître (**pImpl** = **P**ointer of **Impl**ementation).

Intérêt :

- si l'implémentation de la classe transférée demande un temps de compilation important.
- permet d'occulter le fonctionnement d'une classe pour distribuer une bibliothèque.

Exemple :

```
//header file: cat.h
class Cat {
private:
  // externe
  class CatImpl;
  // Handle
  CatImpl *cat_;
public:
  Cat();   // Constr.
  ~Cat(); // Destr.
  Purr();
};
```

```
//CPP file:
#include "cat.h"
class Cat::CatImpl {
  Purr();
  ...
};
// forward (implémentation cachée)
Cat::Cat() { cat_ = new CatImpl; }
Cat::~Cat() { delete cat_; }
Cat::Purr() { cat_->Purr(); }
// implémentation locale
CatImpl::Purr(){ ... }
```

Introduction

Un objet ou une expression `f` est appelable (callable) si on peut écrire `e(args)` où `args` est liste d'arguments séparé par des virgules.

Il existe 4 types d'objets ou expressions appelable : les fonctions, les pointeurs de fonction, les functors, et les lambda-expressions.

Caractéristiques : une lambda-expression (notée λ-*exp*)

- représente une unité de code appelable,
- a un type de retour, une liste de paramètres et un corps,
- peut être définie à l'intérieur d'une fonction,

peut être vu comme une fonction inline sans nom.

Intérêt d'une lambda-expression : définir au vol une fonctionnelle qui doit être passée en paramètre à une autre fonction.

- Il existe de nombreuses fonctions définies dans la stl permettant d'appliquer une fonctionnelle sur des containers.
- En général, une bonne idée d'avoir un ou plusieurs paramètres fonctionnels afin d'appliquer le même algorithme sur des objets complexes

  Déjà dans l'air avec la surcharge d'opérateur + template.

Définition d'une λ-expression

Une lambda-expression est de la forme :

```
[liste-capture] (liste-paramètres) -> type-retour { corps }
```

- `corps` : corps de la lambda-expression.
- `liste-capture` : liste des variables locales définie dans le corps de la λ-*exp*.
- `liste-paramètres` : liste des paramètres passés à la λ-*exp*.
- `-> type-retour` : type de retour de la λ-*exp* (peut être omis, et dans ce cas, le type est déduit de l'argument du `return`).

Exemple : d'une lambda-expression (hors contexte)

```
[](int val) { return 0 < val && val < 10; }
```

Liste de capture :

- `[a,&b]` : capture `a` par valeur, capture `b` par référence.
- `[this]` : capture le pointeur `this` par valeur (dans une méthode).
- `[&]` : capture toutes les variables locales par référence.
- `[=]` : capture toutes les variables locales par valeur.
- `[]` : ne capture rien.
- `[=,&a]` : capture tous les variables locales par valeur, sauf `a` qui est capturé par référence.
- `[&,=a]` : capture tous les variables locales par référence, sauf `a` qui est capturé par valeur.

Définition d'une λ-expression

Attention :

- on ne capture que les variables locales non statiques.
- la liste de capture ne doit pas capturer plusieurs fois le même élément. **exemple :** `[=]` contient `this`, `[&]` contient `&a`, ...
- la capture par un argument d'une entité ne prolonge pas sa durée de vie (ce point est précisé ci-après).
- pour capturer les membres de l'objet courant dans une de ses méthodes, capturer `this`.

Exemple :

```
size_t v1 = 42; // local variable
auto fun1 = [v1] { return v1; };    // passage par copie
auto fun2 = [&v1] { return v1; };   // passage par référence
v1 = 0;
// appels
auto x1 = fun1(); // x1 est 42 (copie de v1)
auto x2 = fun2(); // x2 est 0 (référence vers v1 )
```

Exemples

Exemple d'utilisation avec la stl :

`std::transform` applique un opérateur sur chaque élément d'un intervalle d'un container `[first,last]`, et écrit le résultat à l'emplacement `result` indiqué. L'implémentation est typiquement :

```
template < class InputIterator , class OutputIterator ,
           class UnaryOperator >
   OutputIterator transform(InputIterator first ,
             InputIterator last, OutputIterator result ,
             UnaryOperator op) {
    while (first != last) {
       *result = op(*first);
       ++result;
       ++first;
    }
    return result;
}
```

Utilisation avec une λ-*exp* pour appliquer en place le modulo 2 sur chacun des éléments d'un container :

```
int m = 2;
std::vector<int> v{5, 32, 7, 12, 28};
std::transform(v.begin(), v.end(), v.begin(),
                    [m](int x){ return x % m;});
```

Exemples

Exemples : d'utilisation d'une lambda expression

Pas de capture :

```
vector<string> words;
// initialisation de words ici
// tri de ces mots par taille croissante
stable_sort(words.begin(), words.end(),
   [](const string &a, const string &b)
     { return a.size() < b.size(); });
```

Capture d'un entier (par copie) :

```
size_t   sz = 6;
// retourne un itérateur sur le premier mot de taille
// supérieure où égale à 6
auto wc = find_if(words.begin(), words.end(),
       [sz](const string &a){ return a.size() >= sz; });
```

Calcul le produit des éléments d'un container :

```
std::vector< int > v{2, 3, 4};
int prod = 1;
std::for_each(v.begin(), v.end(),
     [&prod]( int x)-> void {prod *= x;});
// prod contient le résultat
```

Implémentation

Vocabulaire associé :

- une **lambda expression** est juste une expression (partie du code).
- une **fermeture** (closure) est l'objet créé à l'exécution par une λ-*exp*. Cet objet contient les copies ou les références des variables capturées.
- le **classe de la fermeture** (closure class) est la classe (unique pour chaque λ-*exp*) avec laquelle la fermeture est instanciée.

Une λ-*exp* est implémentée sous forme d'un functor avec `operator()` toujours `inline`.

Exemple : Implémentation équivalente avec un functor

```
class cum_prod {
private: int & prod;
public: cum_prod( int & prod_) : prod(prod_) {}
        inline void operator ()(int x) const {prod *= x;}
};
```

```
std::vector< int > v{2, 3, 4};
int prod = 1;
std::for_each(v.begin(), v.end(), cum_prod(prod));
// le functor appelé est cum_prod(prod)(*it)
```

Implémentation

Remarques :

- Le type d'une fermeture ne peut être nommé, mais il peut être déduit avec auto.

  Exemple :
  ```
  auto fun1=[](int i) { return i + 4; };
  ```

- Une λ-*exp* peut être stockée dans un objet `std::function` mais elle ne sera probablement pas `inline` :

  Exemple :
  ```
  std::function<int(int,int)> fun2
          = [=](int a, int b) { return a+2*b; };
  ```

- Une λ-*exp* peut être convertie en un pointeur de fonction.

  Exemple :
  ```
  int& (*fpi)(int*) = [](int* a)->int& { return *a; };
  ```

- La qualification `mutable` d'une λ-*exp* permet au corps de modifier les paramètres capturés par copie, ou appeler leurs méthodes non const.

  Exemple :
  ```
  int count = 5;
  auto get_count = [count]() mutable
       -> int { return count++; };
  int c;
  while ((c = get_count()) < 10) {
          std::cout << c << "\n"; }
  ```

Captures et références

Attention : une λ-*exp* a été conçue pour s'exécuter localement dans l'environnement dans lequel elle est définie :

- dans la mesure du possible, essayer d'utiliser une lambda-expression localement,
- sinon, faire **très attention** aux références invalides (dangling references).

Exemple :

```
auto GetLambda() {
    int Z = 4;
    return [&](int x) { return Z + x; };
    // la lambda fait référence à la variable locale Z
}   // détruite ici
```

```
auto fun = GetLambda();
// ici, Z n'existe plus
int y = fun(4); // utilise une variable locale détruite
```

Captures et références

Conséquence : quand une λ-*exp* n'est pas utilisée localement, il faut :

- éviter de faire de capture par référence,
- éviter d'utiliser des pointeurs (capture par valeur), y compris `this`,
- éviter d'utiliser de champs dans la méthode d'un objet (car leurs résolutions utilise `this`),
- éviter d'utiliser de variable `static const` (pas capturée et équivalent à une référence à la variable `static const`).

Sinon, il faut :

- s'assurer que tous les objets capturés par la λ-*exp* (et qu'elle utilise) existent encore au moment de l'appel,
- **et** que la valeur souhaitée soit celles des objets au moment de l'appel et non de leurs définitions.

Captures et références

Solution partielle : à partir de C^{++}_{14}, il est possible de capturer par valeur avec la syntaxe `[val=exp]` où :

- `val` est une variable définie dans la portée de λ-*exp*,
- `exp` est une expression définie dans la portée de la classe de fermeture (i.e. = le contexte de la définition de la λ-*exp*).

Cette méthode permet de conserver la valeur d'une expression ou d'une valeur pointée telle qu'elle était à la création de la λ-*exp* (λ-capture généralisée).

Exemple :

```
int *p = new int(10);
auto fun1 = [=]() { return *p; };
auto fun2 = [v=*p]() { return v; };
*p = 5;
// ici: l'appel fun1() retourne 10
// ici: l'appel fun2() retourne 5
```

Peut aussi être utilisée pour déplacer un objet dans une λ-*exp*. Exemple :
`auto func = [pw = std::move(pw)] { return pw->isValid(); }`

Note : fonctionnalité implémentée sous VS2015.

Lambda expression

Alternative pour C++11 :

`std::bind` crée un wrapper contenant un appel d'une fonction `f` associé à certains de ses arguments (défini dans `<functional>`).

Exemple :

```
auto fun = [](int x) { return x*x; };
auto bindedfun = std::bind(fun,4);
bindedfun();    // appel fun(4)
```

L'ensemble des arguments de la fonction doit être renseigné, mais l'utiliser de `placeholders` permet de lever cette contrainte (où `_i` = `i`ème argument du wrapper).

Exemple :

```
using namespace std::placeholders;
auto fun = [](int a, int b) { return a*b; };
auto bindedfun = std::bind(fun,_1,4);
bindedfun(2);    // appel fun(2,4)
```

Les arguments de `bind` sont copiés (lvalue) ou déplacé (rvalue), mais jamais passé par référence, sauf si l'argument est wrappé avec `std::ref` (ou `std::cref` pour référence constante).

En conséquence, une λ-capture généralisée peut s'écrire en C++11 (en reprenant l'exemple précédent) :

```
auto fun2 = bind([](int v) { return v; },*p);
```

Conclusion

Résumé :

- la possibilité de faire référence à une *rvalue* permet d'optimiser les constructions ou les copies à partir d'objets temporaires ou destinés à être détruit,
- se rappeler que les références universelles permettent d'écrire des fonctions génériques acceptant en paramètre des lvalues ou des rvalues.
- les lambdas expressions qui permettent de manipuler des fonctionnelles aussi facilement que des variables (fonctions locales temporaires, passage de fonction temporaire à une fonction).

Ajout/Modification :

- Ajout sur la manière de créer un pointeur de fonction `C` à partir d'une lambda-expression.