MEHDIDOU99

Zeste de savoir

Une nouvelle fonctionnalité de C++11 : la sémantique de mouvement

12 août 2019

Table des matières

i

Ce cours est la remise en forme d'un cours écrit il y a maintenant deux ans et demi. Rien n'a changé dans le fond, seule la forme a été retravaillée.
Si vous avez des suggestions d'ajouts, notamment en rapport avec le dernier standard, elles sont les bienvenues.

Bonjour ! Avez-vous entendu parlé de C++11 et des nouvelles fonctionnalités qu'il apporte ? Quelle que soit la réponse, ce cours vous apprendra à utiliser l'une des nouveautés - plus très nouvelle depuis 2011 - de ce standard : la sémantique de mouvement. Alors si vous vous sentez d'attaque, allons ensemble à la découverte de ce concept !

Pour comprendre ce cours vous devrez avoir des notions assez solides en C++, mais il n'y a pas besoin d'être un gourou pour comprendre l'essentiel du tutoriel, même si certains passages peuvent utiliser des concepts (plus ou moins) avancés sans les expliquer, ces explications sortant du cadre de ce cours.

Prérequis	Objectifs
Avoir des bases solides en C++ : connaître les concepts de références et de templates notamment	Découvrir la sémantique de mouvement, avec un peu d'approfondissement

1. La sémantique de mouvement, qu'est-ce que c'est ?

La sémantique de mouvement est, comme dit dans l'introduction, un nouveau concept apporté par le standard C++11, qui permet dans certaines situations de remplacer une copie par une «vampirisation» de l'objet source, généralement lorsque celui-ci est amené à disparaître très prochainement (voir la section : «Les rvalue references : un concept clé»).

Ainsi, l'objet, au lieu d'être copié, va se voir «arracher» ses attributs qui seront transmis à l'objet destination.

C'est pourquoi on parle de sémantique de mouvement : l'objet source, au lieu d'être copié, va en quelque sorte être « déplacé » vers l'objet destination.

1.1. Les rvalue references : un concept clé.

On va maintenant tenter de définir sommairement ce qu'est une *rvalue reference*, car c'est un concept important pour comprendre et utiliser la sémantique de mouvement.

1.1.1. Quelques définitions

Pour tenter d'approcher le concept de *rvalue reference*, on va passer par quelques autres définitions, qui permettront d'y venir.

lvalue : Acronyme de left hand side value, c'est-à dire une expression qui peut se trouver à gauche d'un opérateur d'affectation. C'est maintenant généralisé en tout ce qui réfère à un espace en mémoire, donc tout ce dont on peut obtenir l'adresse avec l'opérateur '&'.

rvalue : Acronyme de right hand side value, c'est-à-dire une expression qui peut être à droite d'un opérateur d'affectation. C'est maintenant généralisé en tout ce qui n'est pas une *lvalue*. C'est souvent une variable temporaire (retours de fonctions qui renvoient par valeur...).

rvalue reference : une *rvalue reference* est, je vous le donne en mille, une référence sur une *rvalue*. Une référence classique est, comme vous le savez sûrement, représentée dans le code par le symbole `&`, tandis qu'une *rvalue reference* est symbolisée par `&&`.

Eh oui, tout ça pour ça. Bon. Supposant que vous restez sur votre faim, je vais m'expliquer un peu : en fait, c'est un nouveau concept introduit par C++11 qui permet d'informer le compilateur que la variable est temporaire, donc que l'on peut y appliquer la sémantique de mouvement.

Bon, c'est gentil la théorie, mais nous, on aimerait bien coder un peu, et voir ce que ça fait et comment ça marche en pratique !

Ne vous inquiétez pas, ça arrive bientôt. Je vais juste conclure cette partie en dictant une règle quasi-absolue lorsque l'on utilise la sémantique de mouvement :

Règle quasi-absolue lorsque l'on utilise la sémantique de mouvement : Il ne faut JAMAIS réutiliser un objet qui a été la source d'un mouvement. En effet, il s'est fait vampiriser ses états, et, s'il est sûr qu'il est toujours destructible, rien ne garantit qu'il est dans un état utilisable. Et même du point de vue logique, sémantique, ce serait absurde !

Bon, si vous m'avez écouté jusqu'ici, vous l'avez bien mérité : c'est parti pour un peu de pratique !

2. Le mouvement en pratique

2.1. Rendre un type déplaçable

Lorsque vous créez une classe, elle est copiable et déplaçable par défaut, et vous n'avez pas besoin d'implémenter vos propres constructeur et opérateur d'affectation sauf pour des besoins spécifiques. Cependant, votre compilateur ne créera pas de constructeur et d'opérateur d'affectation par déplacement si vous implémentez ceux par copie vous-même. Qu'à cela ne tienne, nous allons créer nous-mêmes notre constructeur et notre opérateur d'affectation !

J'utiliserai pour cette partie la classe suivante, que l'on va rendre déplaçable :

```
1  class X
2  {
3  public:
4      X()
5      : m_ptr{new int}
6      , m_phrase{"Hello world !"}
7      {
8      }
9
10     X(X const& autre)
11     : m_ptr{new int{*(autre.m_ptr)}}
12     , m_phrase{autre.m_phrase}
13     {
14     }
15
16     X& operator=(X const& autre)
17     {
18         X{autre}.swap(*this);
19
20         return *this;
21     }
22
23     void swap(X& autre)
24     {
25         std::swap(m_ptr, autre.m_ptr);
26         std::swap(m_phrase, autre.m_phrase);
27     }
```

```
28
29 private:
30     int* m_ptr;
31     std::string m_phrase;
32 };
```

J'ai utilisé ici un pointeur nu pour l'exemple, mais dans un vrai projet on utiliserait si possible une référence, et si une référence ne convient pas on utiliserait un pointeur intelligent. Donc ne faites pas ça chez vous, programmez de manière sécurisée

On va donc ajouter à cette classe la possibilité d'utiliser la sémantique de mouvement. Bon, commençons à coder ! Je vous conseille de réaliser les codes d'exemple chez vous pour bien comprendre.

2.1.1. Le constructeur

2.1.1.1. Le prototype Le constructeur par mouvement prend pour seul argument une *rvalue reference* sur l'objet source. Essayez de le deviner tout seul ! C'est facile, j'ai déjà tout dit.

Voilà la solution pour ceux qui dormaient au fond pendant la partie théorique :

 Contenu masqué n°1

C'était dur, hein ?

2.1.1.2. L'implémentation Passons maintenant à l'implémentation :

```
1 X::X(X&& autre) noexcept
2 : m_ptr{std::exchange(autre.m_ptr, nullptr)}
3 // std::exchange remplace la valeur de son premier paramètre par la
4 // valeur du second paramètre et renvoie l'ancienne valeur du
      premier.
5 , m_phrase{std::move(autre.m_phrase)}
6 {
7 }
```

Parenthèse : Si vous comparez l'opérateur d'affectation par déplacement avec celui par copie, on a une allocation dynamique et une copie en moins, ce qui montre déjà qu'il y a moyen d'optimiser son code grâce au mouvement. Ne cherchez pas à comprendre tout de suite la deuxième ligne, qui utilise une fonction que l'on verra plus tard dans ce tutoriel. En attendant, sachez que la ligne de code fait un mouvement de `autre.m_phrase` vers `m_phrase`. Cela permet de remplacer une copie de chaînes par une opération ne coûtant guère plus que la copie d'un pointeur. Dans cet exemple, c'est la principale source de gains en performances.

2.1.2. L'opérateur d'affectation

2.1.2.1. Le prototype L'opérateur d'affectation par déplacement prend une *rvalue reference* en paramètre et renvoie une *lvalue reference* (une référence "classique"). Si vous avez suivi, vous devriez trouver tout seul.

Bon, je vous donne la solution :

> Contenu masqué n°2

Bravo si vous avez trouvé vous-mêmes !

2.1.2.2. L'implémentation L'implémentation est très facile. En effet, on va utiliser le constructeur par mouvement, pour respecter un idiome que l'on pourrait appeler *move-and-swap* (équivalent de l'idiome *copy-and-swap*, mais pour le mouvement) :

> Contenu masqué n°3

Voilà ! Vous avez un type déplaçable !

? Cool ! J'ai fait mon type déplaçable ! Maintenant, j'aimerais bien savoir comment l'utiliser vraiment !

Ne vous impatientez pas, ça vient ! Mais avant, je vais vous montrer une astuce bien pratique.

2.1.3. Petite astuce pour les fainéants

Imaginez que vous ayez une classe dans laquelle vous voulez implémenter vos propres constructeurs et opérateur d'affectation par copie, mais que pour ce qui est du mouvement, ceux par défaut vous conviennent. Mais si vous implémentez ceux par copie, le compilateur ne va pas les créer pour vous. On aimerait donc le forcer à le faire ; ceci est très simple, et se fait avec la syntaxe suivante :

```
1  class X
2  {
3      // tout plein de trucs...
4      X(X const& autre)
5      {
6          // votre implémentation
7      }
8
9      X& operator=(X const& autre)
10     {
```

```
11          // votre implémentation
12      }
13
14      X(X&&) = default;
15      X& operator=(X&&) = default;
16  }
```

Voilà, tout simplement !

En fait, dès qu'une ou plusieurs des cinq fonctions que sont les constructeurs et opérateurs d'affectation par copie et mouvement et le destructeur sont implémentées par le développeur, les autres devraient être créées par défaut de manière explicite (règle du tout ou rien, voir cet article pour plus de détails).

2.2. Utiliser un type déplaçable

À la question «Comment utiliser le mouvement ?», je répondrais... laisser faire le compilateur ! Eh oui ! Le compilateur sait très bien faire les optimisations incluant l'utilisation du mouvement ! Par exemple, lorsque vous renvoyez un objet déplaçable par valeur dans une fonction, le compilateur optimisera tout seul selon les cas de figure, et saura si oui ou non il faut renvoyer une *rvalue reference* pour utiliser le mouvement !

On verra quelques exemples d'optimisations par le compilateur dans une section ultérieure. Pour l'instant, on va voir comment utiliser explicitement l'utilisation du mouvement.

2.2.1. Forcer l'utilisation du mouvement

Il se peut parfois dans un code que vous vouliez vous-mêmes utiliser le mouvement, au-delà des optimisations du compilateur. D'ailleurs vous avez déjà vu une telle utilisation du mouvement, dans l'implémentation de l'opérateur d'affectation par déplacement lui-même. Souvenez-vous, c'était la ligne "secrète" :

```
1  m_phrase = std::move(autre.m_phrase);
```

Eh bien, c'est le moment d'y revenir ! En fait, la fonction utilisée peut être assimilée à `static_cast<std::string&&>(autre.m_phrase)`. Ainsi, la fonction convertit un objet (ou une référence) en *rvalue reference*, pour pouvoir y appliquer le mouvement !

Si vous utilisez `std::move`, soyez sûrs d'avoir de bonnes raisons de le faire. Par exemple, évitez de renvoyer des *rvalue references* en retour de fonctions : préférez le renvoi par valeur, le compilateur saura optimiser beaucoup mieux que vous ! Si vous vous demandez pourquoi, tapez « copy elision » dans un moteur de recherche.

A noter que std : :move porte en fait très mal son nom. En effet, il ne réalise aucun mouvement. Ainsi, `T&& ref = std::move(variable)` ne fait aucun mouvement, cela crée juste une *rvalue reference* vers variable.

3. Quelques exemples d'utilisations du mouvement.

Dans cet extrait, nous allons aborder, en vrac, des exemples qui nous permettront de comprendre ce que la sémantique de mouvement permet de réaliser, notamment en termes de gains de performances.

3.1. Sommaire

— std : :vector : des performances améliorées (exercice inside).
— std : :unique_ptr : le transfert de l'ownership.

3.2. std : :vector : des performances améliorées

Lorsque vous voulez ajouter un élément au début d'un vector d'éléments copiables, voici les étapes qui se déroulent :

1. Un nouveau tableau est alloué, avec une taille supérieure de 1 par rapport au tableau sous-jacent actuel du vector.
2. Tous les éléments du vector sont copiés dans ce nouveau tableau.
3. L'élément à ajouter est construit puis copié au dernier emplacement restant.

Comme vous voyez, ça fait beaucoup de copies d'éléments ! Alors qu'avec un élément déplaçable, std : :vector va préférer le mouvement à la copie. Autant vous dire que si le vector est assez gros, cela peut représenter un gros gain en performances, qui vaut largement le temps de développement des constructeur et opérateur d'affectation, surtout quand le compilateur peut les écrire à notre place !

A noter que cette optimisation fonctionne que vous utilisiez `push_back` ou `emplace_back`.

Même avec l'utilisation de la sémantique de mouvement, la copie d'un vector dans son intégralité peut s'avérer particulièrement coûteuse. Donc si vous savez que vous allez avoir à ajouter de nombreux éléments dans un vector, pensez à utiliser std : :vector : :reserve().

3. Quelques exemples d'utilisations du mouvement.

3.2.0.1. Parenthèse : std : :move_if_noexcept

En fait, std : :vector n'utilisera le mouvement que si les constructeur et opérateur d'affectation par mouvement garantissent qu'ils ne lancent pas d'exceptions (avec le mot-clé noexcept). En effet, imaginons que le vector en question ait déplacé la moitié de ses éléments puis qu'une exception est lancée. Alors, il se retrouve dans un état incohérent, puisque la moitié des éléments ont été déplacés, mais pas l'autre moitié. Alors qu'avec la copie, std : :vector n'a pas besoin de cette même garantie, puisque si une exception est lancée, le vector peut rester sur son état initial (les éléments copiés étant encore dans un état cohérent).

Pour cela, std : :vector utilise la fonction std : :move_if_noexcept, qui renvoie une *rvalue reference* vers son paramètre si celui-ci dispose de constructeur et opérateur d'affectation par mouvement noexcept, ou renvoie l'objet lui même si cette condition n'est pas vérifiée.

3.2.1. Exercice

Essayez de coder une fonction qui prend en argument un vector et un itérateur vers un élément de ce vector, et qui déplace cet élément à la fin du vector. Il ne faut aucune copie, que du mouvement.

3.2.1.1. Solution

Contenu masqué n°4

3.3. std : :unique_ptr : le tranfert de l'ownership

`std::unique_ptr` est un pointeur intelligent qui est fait pour être l'unique responsable de la ressource sur laquelle il pointe. Ainsi, dès qu'il sort de sa portée, il détruit cette ressource. Cependant, un problème se pose lorsque l'on souhaite tranférer cette responsabilité : si l'on copie le `std::unique_ptr`, les deux `std::unique_ptr` pointeront vers la même ressource. Et la, c'est la catastrophe : double libération de la mémoire, ou déréférencement d'un des deux pointeurs lorsque l'autre a déjà détruit la ressource. La copie de `std::unique_ptr` est donc interdite :

```
1 // dans la classe std::unique_ptr
2 unique_ptr(unique_ptr const&) = delete;
3 unique_ptr& operator=(unique_ptr const&) = delete;
```

La sémantique de mouvement vient alors à notre rescousse, et permet de transférer l'*ownership* vers le nouveau pointeur ; l'ancien perdant l'*ownership*, il n'y a plus de problème de double délétion !

Selon la règle absolue de la sémantique de mouvement, vous ne devez pas réutiliser le pointeur source.

4. [Bonus] Exemples d'utilisations des rvalue references autres que le mouvement

Dans cette section, on va étudier quelques exemples d'utilisation des rvalue references qui ne concernent pas le mouvement.

4.1. Sommaire

— Un exemple amusant : les fonctions ref-qualifiées
— Une fonctionnalité très intéressante : le perfect forwarding

4.2. Un exemple amusant : les fonctions ref-qualifiées

Les fonctions ref-qualifiées sont un cas particulier de surcharge de fonction membre qui permet d'obtenir un comportement polymorphique selon si l'objet sur lequel est appelée la fonction (autrement dit `*this`) est une lvalue reference ou une rvalue reference. Leur fonctionnement (ainsi que leur syntaxe) ressemble à celui des fonctions const-qualified. Vous avez sûrement déjà vu ce type de code :

```
class Y
{
    // du code...
    void foo();

    void foo() const;
    // encore du code...
};
```

Ce code permet de surcharger la fonction Y : :foo() pour les objets constants. *Eh* bien on peut faire la même chose pour la surcharger avec les références ! Ainsi, on peut créer une fonction membre qui se comporte différemment selon si elle est appliquée à une *lvalue reference* (ou tout ce qui peut s'y ramener) ou une *rvalue reference* (ou tout ce qui peut s'y ramener) :

```
class Z
{
public:
    Z() = default;

    void foo() &
    {
        std::cout << "Je suis un Z& !" << std::endl;
    }

    void foo() &&
```

4. [Bonus] Exemples d'utilisations des rvalue references autres que le mouvement

```
    {
        std::cout << "Je suis un Z&& !" << std::endl;
    }
};
```

On peut même aller plus loin et écrire une surcharge pour les références constantes :

```
class Z
{
public:
    Z() = default;

    void foo() &
    {
        std::cout << "Je suis un Z& !" << std::endl;
    }

    void foo() &&
    {
        std::cout << "Je suis un Z&& !" << std::endl;
    }

    void foo() const&
    {
        std::cout << "Je suis un const Z& !" << std::endl;
    }

    void foo() const&&
    {
        std::cout << "Je suis un const Z&& !" << std::endl;
    }
};

// Testons :

int main()
{
    Z unZ{};
    unZ.foo();
    std::move(unZ).foo();

    Z().foo();

    const Z myConstZ{};
    myConstZ.foo();
    std::move(myConstZ).foo();

    return 0;
```

```
42 }
```

La sortie sera :

```
1 Je suis un Z& !
2 Je suis un Z&& !
3 Je suis un Z&& !
4 Je suis un const Z& !
5 Je suis un const Z&& !
```

À vous de faire une multitude de tests si vous le souhaitez

À noter : si on exclut les surcharges avec le qualificateur volatile (que vous pouvez ajouter si ça vous amuse), on ne peut pas faire d'autres surcharges ; votre compilateur râlerait à cause de l'ambiguïté qui serait occasionnée lors du choix de la bonne surcharge. Ainsi, si vous ajoutez à l'exemple précédent une surcharge sans qualificateur de référence, vous aurez plusieurs erreurs (ressemblant à "error : void foo() & cannot be overloaded with void foo()"), même si cette surcharge est qualifiée const et/ou volatile.

Cette fonctionnalité amusante peut être utilisée à des fins d'optimisation, par exemple.

4.3. Une fonctionnalité très intéressante : le perfect forwarding

Cette partie est fortement inspirée de la partie sur le perfect forwarding de l'article de T. Becker sur les *rvalue references*.

Le *perfect forwarding* est une technique très utile, notamment lorsqu'on programme avec des templates. En effet, on aura parfois besoin de faire passer tels quels les arguments d'une fonction à une autre fonction utilisée dans le corps de la première. Le *perfect forwarding* permet de le faire, comme son nom l'indique, parfaitement.

4.3.1. Exemple : une factory

Un exemple typique où l'on a besoin de cette technique est une *factory* (un objet dont le rôle est de construire d'autres objets) template, comme `std::make_unique`. Une première approche serait :

```
1 template<typename T, typename ... Args>
2 std::unique_ptr<T> make_unique(Args ... args)
3 {
4     return std::unique_ptr<T>{new T{args...}};
5 }
```

4. [Bonus] Exemples d'utilisations des rvalue references autres que le mouvement

Le problème, c'est que si le constructeur prend son argument par référence, il y aura une copie inutile (voire nuisible, voire interdite !) de l'élément. Ainsi, on n'a pas le comportement voulu, à savoir une fonction qui fait comme si le constructeur était appelé directement, sans que cette indirection supplémentaire ait d'impact. On pourrait alors y remédier comme ça :

```
template<typename T, typename ... Args>
std::unique_ptr<T> make_unique(Args& ... args)
{
    return std::unique_ptr{new T{args...}};
}
```

Mais il reste toujours un problème. En effet, regardez ce qui se passe si on fournit à cette implémentation de `make_unique` un paramètre de cette manière :

```
auto ptr = make_unique<UnType>(foo())
```

Ce code ne fonctionnera pas : en effet le retour de `foo()` est une *rvalue*, qui ne peut pas être convertie en *lvalue reference*. On peut alors ajouter un `const`, mais ce ne sera pas viable car tous les paramètres ne sont pas forcément voués à être constants. De plus, avec une référence constante, on ne peut pas appliquer de sémantique de mouvement, même si cela aurait été possible en passant directement par le constructeur. On n'a donc toujours pas de perfect forwarding.

Heureusement, les *rvalue references* vont encore venir à notre rescousse !

En effet, elles peuvent se comporter comme des références universelles dans les fonctions templates : si une fonction `template<typename T> void foo(T&&)` est appelée sur une lvalue, alors le paramètre sera considéré comme une lvalue reference T&. Si elle est appelée sur une rvalue (un retour de fonction par exemple), alors le paramètre restera une rvalue reference.

De plus, lorsque dans une déduction template on se retrouve avec des types étant marqués plusieurs fois par des références, celles-ci sont simplifiées selon des règles :

Référence du paramètre template	**Référence qui s'y ajoute**	**Résultat**
&	&	&
&	&&	&
&&	&	&
&&	&&	&&

Tout cela est utilisé par la fonction std : :forward, qui permet de "choisir" la bonne référence selon si ce qu'on lui donne en paramètre est une lvalue ou une rvalue. On peut la définir ainsi :

```cpp
template<class T>
T&& forward(typename std::remove_reference<T>::type& x) noexcept
{
    return static_cast<T&&>(x);
}
```

Ainsi, si elle est appelée sur une lvalue de type A, le paramètre sera évalué comme A&, et on obtient :

```cpp
A& && forward(typename std::remove_reference<T>::type& x) noexcept
{
    return static_cast<A& &&>(x);
}
```

Puis remove_reference est évalué et les références se simplifient :

```cpp
A& forward(A& x) noexcept
{
    return static_cast<A&>(x);
}
```

On obtient donc bien, à la fin une *lvalue reference*. De même, si on applique le même processus avec une *rvalue*, on obtient une *rvalue reference*. La fonction std : :forward permet donc bien de *forwarder* parfaitement un argument.

Si l'on transforme notre factory pour utiliser ce que l'on vient de voir, on obtient :

```cpp
template<typename T, typename ... Args>
std::unique_ptr<T> make_unique(Args&& ... args)
{
    return std::unique_ptr<T>{new T{std::forward<Args>(args)...}};
}
```

Les mêmes processus s'appliquent (déduction des paramètres templates et simplification des références), vous pouvez essayer de voir ce qui se passe étape par étape. Je ne vais pas le faire ici, pour ne pas faire redondance avec l'explication du fonctionnement de `std::forward`, et éviter que ce soit trop long.

Remarquez juste l'expansion du pack de paramètres : on applique `std::forward` à chaque paramètre séparément et on étend le pack ensuite.

On arrive à la fin de ce tutoriel. J'espère qu'il vous aura plu, et que j'ai réussi à vous apprendre quelques notions intéressantes

J'ajoute, comme promis, le lien vers le cours de Thomas Becker : Rvalue references explained (en anglais).

Un autre lien intéressant est une page de documentation sur les différents types de valeurs . En effet, il y a d'autres types de valeurs que simplement les lvalue et les rvalue ; cette page fait donc la liste de tous les types de valeurs existant en C++.

J'aimerais finir en remerciant ceux qui m'ont aidé à compléter ou corriger le cours, notamment jo_link_noir, lmghs et Freedom, ainsi qu'Arius, pour son premier retour, et informaticienzero, mon validateur préféré

Ceci est mon premier cours, je suis donc ouvert à toute remarque pouvant servir à l'améliorer (ou à m'améliorer). Merci à tous et à bientôt !

Contenu masqué

Contenu masqué n°1

```
X::X(X&& autre) noexcept;
// si vous êtes sûr qu'il ne peut pas lancer une exception,
    spécifiez-le
```

Retourner au texte.

Contenu masqué n°2

```
X& X::operator=(X&& autre) noexcept;
```

Retourner au texte.

Contenu masqué n°3

```
X& X::operator=(X&& autre) noexcept
{
   X{std::move(autre)}.swap(*this);

   return *this;
```

```
6 }
```

Retourner au texte.

Contenu masqué n°4

```
1  template<typename T>
2  typename std::vector<T>::iterator deplacerALaFin
3     (std::vector<T>& collection, typename std::vector<T>::iterator
          pos)
4  {
5     if(pos + 1 != collection.end())
6     {
7         T aDeplacer{std::move(*pos)};
8         for(; pos != collection.end() - 1; ++pos)
9             *pos = std::move(*(pos + 1));
10
11        *pos = std::move(aDeplacer);
12    }
13
14    return pos;
15 }
```

Retourner au texte.