

Ch. 10 - Sémantique de mouvement

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique



9 décembre 2020

Table des matières

- 1 Introduction
- 2 Références de rvalue
- 3 Constructeur et affectation de mouvement
- 4 La sémantique de mouvement
- 5 Idiomes
 - Copy and swap
 - RAII

Introduction

Le besoin de performances

- Par défaut, les paramètres sont passés par valeur en C++

Problème

- Performances en cas de copie de gros objets

Solution

- Utiliser le passage par référence
 - Utiliser le passage par adresse
-
- Parfois, cette solution ne suffit pas
 - Création implicite d'objets temporaires

Le problème des temporaires

Exemple de fonction

```
■ string hello() { return string("Hello"); }
```

- Cette fonction retourne un `string` par valeur
- À chaque appel, la fonction
 - 1 Crée un temporaire pour y stocker le littéral `"Hello"`
 - 2 Retourne une copie de ce temporaire
- Ça fait beaucoup de temporaires...
- Ce problème se pose avec toutes les fonctions retournant un gros objet construit localement
- Solution médiocre : allouer dynamiquement (avec `shared_ptr`)
 - Plus lent qu'une allocation automatique
- Excellente solution : utiliser la sémantique de mouvement
 - Mis en œuvre par le biais des références de rvalue

Références de rvalue

Le besoin de lvalues

- Analysons `int x = 42;`
 - 42 est une rvalue (un littéral)
 - 42 n'a pas d'adresse
 - On le stocke dans une lvalue pour l'utiliser

- Analysons

```
string getString() {  
    return "hello_world";  
}  
  
string s4 = getString();
```

- Même cas que précédemment
 - ... mais on utilise un littéral après un appel de fonction
 - On le stocke dans une lvalue pour l'utiliser

Le cas des temporaires

- Analysons `int y = x + 1;`
 - `x + 1` est une rvalue
 - Le compilateur crée un objet temporaire pour stocker le résultat de l'opérateur `+`
 - On stocke le temporaire rvalue dans une lvalue
- Analysons

```
string s1 = "hello ";  
string s2 = "world";  
string s3 = s1 + s2;
```

- Même cas que précédemment
- La gestion des temporaires peut coûter des ressources
 - Surtout si les temporaires créés sont nombreux / gros

Références de rvalue

- Référence conçue pour être initialisée avec une rvalue seulement

Syntaxe

- On utilise l'opérateur `&&`
- `int && x = 5;`
- `auto && f {Fraction(2,3)};`
- Permet d'étendre la durée de vie d'un temporaire
- Par le biais de la sémantique de déplacement, elles permettent d'éviter des copies d'objets

Exemple

■ Fichier rvalue-ref.cpp

```
1  int main()
2  {
3      string s1 = "Hello_";
4      string s2 = "World";
5
6      string && s3 = s1 + s2; //rvalue reference
7      cout << s3 << endl;;
8
9      s3 += "!";
10     sout << s3 << endl;
11 }
```

Règles d'appel

■ Fichier `surdef-rvalue-ref.cpp`

```
1 void f(int & lref) // l-value arguments will select this function
2 {
3     cout << "l-value_reference" << endl;
4 }
5
6 void f(const int & lref) // const l-value arguments will select this function
7 {
8     cout << "l-value_reference_to_const" << endl;
9 }
10
11 void f(int && rref) // r-value arguments will select this function
12 {
13     cout << "r-value_reference" << endl;
14 }
15
16 int main()
17 {
18     int x{ 5 };
19     f(x); // l-value argument calls l-value version of function
20     f(5); // r-value argument calls r-value version of function
21 }
```

Création de références

- On ne peut pas créer de références à partir de n'importe quoi
- Le tableau ci-dessus illustre les créations possibles

	lvalue	const lvalue	rvalue	const rvalue
T&	oui			
const T&	oui	oui	oui	oui
T&&			oui	
const T&&				oui

- En pratique, `const T&&` n'est jamais utilisé
 - Aucun intérêt de « lire depuis un temporaire »

Forwarding arguments

- En C++, étant donné une expression $E(a_1, a_2, \dots, a_n)$, il n'est pas possible d'écrire une fonction f telle $f(a_1, a_2, \dots, a_n)$ soit équivalent
 - ... pour des raisons techniques
- On ne peut pas écrire `void f1(int && i) {}` et `void f2(int&& i) { f1(i); }`
- Ce problème est appelé le « forwarding problem »
- Avant les références de rvalue, plusieurs solutions non satisfaisantes existaient
- Solution : utiliser `std::forward`
 - `void f2(int&& i) { f1(std::forward<int>(i)); }`
- Sucre syntaxique pour `static_cast<T&&>(i)`
- Plus de détails au Ch. 13

Constructeur et affectation de mouvement

Permettre la sémantique de mouvement

Objectif

- On veut éviter au maximum la copie de temporaires
- On évite cela en utilisant la sémantique de mouvement
 - Parfois appelé sémantique de déplacement
- Mis en œuvre à l'aide de deux outils
 - 1 Constructeur de mouvement
 - 2 Opérateur d'affectation de mouvement
 - Par opposition à opérateur d'affectation de copie
- Constructeur et opérateur appelés selon les règles d'appel de fonctions
 - Sélectionnés si appelé avec des rvalue

Constructeur de mouvement

■ Syntaxe

- `T(T&& t)`
- `T(T&& t) = delete`
- `T(T&& t) = default`

■ Constructeur de mouvement implicitement généré si toutes les conditions suivantes sont vraies

- Il n'y a ni constructeur de copie ni opérateur d'affectation de copie explicite
- Il n'y a ni constructeur de mouvement ni opérateur d'affectation de mouvement explicite
- Il n'y a pas de destructeur explicite

■ Constructeur de mouvement implicitement = `delete` si au moins une des conditions suivantes est vraie

- `T` a des attributs non statiques qui ne peuvent pas être déplacés
- `T` a une superclasse qui ne peut pas être déplacée ou détruite

Opérateur d'affectation de mouvement

■ Syntaxe

- `T& operator=(T&& t)`
- `T& operator=(T&& t) = delete`
- `T& operator=(T&& t) = default`

■ Opérateur implicitement généré si toutes les conditions suivantes sont vraies

- Il n'y a ni constructeur de copie ni opérateur d'affectation de copie explicite
- Il n'y a ni constructeur de mouvement ni opérateur d'affectation de mouvement explicite
- Il n'y a pas de destructeur explicite

■ Constructeur de mouvement implicitement = `delete` si au moins une des conditions suivantes est vraie

- `T` a des attributs non statiques qui ne peuvent pas être déplacés
- `T` a une superclasse qui ne peut pas être déplacée ou détruite

Exemple (1/2)

■ Fichier `move.cpp`

```
1 struct A
2 {
3     int i;
4
5     A(int i = 0) : i(i) { cout << "+" << endl; } //default cstr
6     A(const A& a) : i(a.i) { cout << "c" << endl; } //copy cstr
7     A(A&& a) : i(std::move(a.i)) { cout << "m" << endl; } //move cstr
8     A& operator=(const A& a)
9     {
10         cout << "=c" << endl;
11         if (this != &a)
12             i = a.i;
13         return *this;
14     }
15     A& operator=(A&& a)
16     {
17         cout << "=m" << endl;
18         i = std::move(a.i);
19         return *this;
20     }
21 };
```

■ Désactiver optimisations `gcc : -fno-elide-constructors`

Exemple (2/2)

■ Fichier `move.cpp`

```
1 //void f(A a) { cout << "by value" << endl; } //1
2 void f(A& a) { cout << "by_ref" << endl; } //2
3 void f(const A& a) { cout << "by_const_ref" << endl; } //2
4 void f(A&& a) { cout << "by_rvalue_ref" << endl; } //2
5
6 int main()
7 {
8     A a1(1);
9     f(a1);
10
11     const A a2(2);
12     f(a2);
13
14     f(A(3));
15 }
```

■ Désactiver optimisations gcc : `-fno-elide-constructors`

Pourquoi tant de haine ?

Question

- Pourquoi implémenter des constructeurs de mouvement si le compilateur optimise ?
- En pratique, l'une des seules optimisations effectuée est « Return value optimisation »
 - Uniquement sur le retour de fonction
- Pas d'application sur les paramètres de fonctions
- En particulier, le standard prend régulièrement des références de rvalue en paramètre
- Si le constructeur de mouvement et l'opérateur d'assignation de mouvement sont implémentés, offre de bonnes optimisations
- `std::move` permet de « déplacer » une rvalue
 - En pratique, convertit une lvalue en rvalue
 - Ne crée pas de temporaire

Erreur courante

■ Ne faites pas ça

```
1 A create_A()  
2 {  
3     A a;  
4     return std::move(a);  
5 }
```

■ Et *surtout pas* ça

```
1 A&& create_A()  
2 {  
3     A a;  
4     return std::move(a);  
5 }
```

■ Les déplacements sont exclusivement effectués par le constructeur de déplacement (implicitement)

La sémantique de mouvement

Hygiène

- En pratique, dès qu'une classe « gère une ressource », elle *doit* au moins implémenter
 - 1 un destructeur
 - 2 un constructeur de copie
 - 3 un opérateur d'affectation de copie
- Si l'on a besoin de l'un, on a besoin des trois
- Pour des raisons d'optimisation, on veut parfois
 - 1 un constructeur de mouvement
 - 2 un opérateur d'affectation de mouvement
- Ressource : wrapper, mémoire dynamique, pointeur, mutex, etc.
- Critères de qualité
 - Éviter la duplication de code
 - Garantie forte d'exception

Rules of three / five / zero

La règle des trois

- Si une classe a besoin d'un destructeur, d'un constructeur de copie ou d'un opérateur d'affectation de copie, elle a besoin des trois

La règle des cinq

- Si une classe a besoin d'un constructeur de mouvement ou d'un opérateur d'affectation de mouvement, elle a besoin
 - de la règle des trois
 - d'un constructeur de mouvement *et* d'un opérateur d'affectation de mouvement
- L'idiôme « copy and swap » permet de fournir des critères de qualité

Dans la plupart des cas

- Dans la plupart des cas, les classes ne « gèrent pas de ressources »
 - Pas de wrappers, de pointeurs (intelligents ou non), de mutex, etc.
- Dans ce cas, il n'est pas nécessaire d'implémenter
 - la règle des trois
 - la règle des cinq
- Les sémantiques de copie et de déplacement sont implémentées correctement par défaut
- Si une classe ne bénéficie pas de la sémantique de mouvement, il n'est pas non plus nécessaire de l'implémenter
- Le principe POO de responsabilité unique implique également qu'une classe gérant une ressource doivent *uniquement* s'occuper de cette gestion
 - « Règle de zéro » pour les autres classes

Construction d'un exemple académique

- On veut créer une classe modélisant un tableau à taille statique d'entiers
 - Sans utiliser les conteneurs, qui implémentent la règle des cinq
- Fonctionnalités :
 - un constructeur, qui alloue la mémoire dynamiquement
 - un opérateur `[]`, qui permet l'accès et l'affectation
- Cette classe est de taille « arbitrairement grande »
 - On parle ici de l'espace total alloué, pas du résultat de `sizeof`

La base

■ Fichier `static-array.cpp`

```
1  class Array
2  {
3      unsigned _size;
4      int * data;
5
6      public:
7          Array(unsigned size = 0)
8              : _size(size),
9              data(size != 0 ? new int[size] : nullptr)
10         {}
11
12         int& operator[](unsigned pos)
13         {
14             return data[pos];
15         }
16
17         int size() const
18         {
19             return _size;
20         }
21
22         ...
23     };
```

Debriefing d'analyse

Constatations

- On veut sûrement passer le plus souvent possible `Array` par référence
 - Classe assez volumineuse
 - Si on compte les données allouées avec `new`
 - Il faut implémenter la règle des trois si l'on veut
 - que la copie et l'affectation recopie soient fonctionnelles
 - éviter les fuites mémoires
 - Il faut implémenter la règle des cinq si l'on veut éviter des copies de temporaires inutiles
-
- Pour des raisons académiques, on implémentera une fonction `create_increasing_array(n)`, qui crée un tableau d'éléments `[0, ..., n-1]`

La règle des trois

■ Fichier `static-array.cpp`

```
1 ~Array()
2 {
3     if (data)
4         delete[] data;
5 }
6
7 Array& operator=(const Array& a)
8 {
9     if (this != &a)
10    {
11        if (data)
12            delete[] data;
13
14        _size = a._size;
15        data = a._size != 0 ? new int[a._size] : nullptr;
16        std::copy(a.data, a.data + _size, data);
17    }
18
19    return *this;
20 }
21
22 Array(const Array& a) : _size(a._size), data(a._size != 0 ? new int[a._size] : nullptr)
23 {
24     std::copy(a.data, a.data + _size, data);
25 }
```

La règle des cinq

■ Fichier `static-array.cpp`

```
1  Array(Array && a) :  
2      _size(std::move(a._size)),  
3      _data(std::move(a._data))  
4  {  
5      a._data = nullptr; //ask why I should do this  
6  }  
7  
8  Array& operator=(Array && a)  
9  {  
10     _size = std::move(a._size);  
11     _data = std::move(a._data);  
12     a._data = nullptr;  
13  
14     return *this;  
15 }
```

- Bonne pratique : utiliser `std::move` dans l'implémentation de la sémantique de déplacement

Debriefing de conception

Problème

- Cela fait *vraiment* beaucoup de copier / coller
- Il faut trouver une solution

Idiomes

C++ et les idiomes

- C++ est un langage qui requiert une bonne hygiène de programmation
- Une bonne partie de cette hygiène est implémentée via des « idiomes »
- Idioms : forme de conception d'un fragment de code afin d'y intégrer des critères de qualité

Objectif courant

- Implémenter la règle des cinq et éviter le copier / coller
- On requiert souvent les constructeurs de déplacement à être `noexcept`
 - Le standard choisit souvent le constructeur de recopie si le constructeur de déplacement n'est pas `noexcept`

La clé

- Si on a besoin d'implémenter la règle des cinq, l'astuce réside dans
 - 1 l'implémentation d'une fonction `swap` (souvent amie)
 - 2 le passage par valeur du paramètre de l'opérateur d'affectation de recopie

Hygiène C++

- Si une copie doit être faite, faites-la grâce au passage par valeur plutôt que manuellement
 - Offre des optimisations compilatoires
-
- Hormis l'implémentation de la règle des cinq, le reste de la classe ne change pas

Illustration

■ Fichier `static-array-copy-and-swap.cpp`

```
1 ~Array ()
2 {
3     if (data)
4         delete [] data;
5 }
6
7 friend void swap(Array& a1, Array& a2) noexcept
8 {
9     std::swap(a1.size, a2.size);
10    std::swap(a1.data, a2.data);
11 }
```

■ Bonne pratiques

- 1 Utiliser `std::swap` dans l'implémentation d'une fonction `swap`
- 2 Rendre `swap` `noexcept` pour l'utiliser dans le constructeur de déplacement

Illustration

■ Fichier `static-array-copy-and-swap.cpp`

```
1  Array& operator=(Array a)
2  {
3      swap(*this, a);
4
5      return *this;
6  }
7
8  Array(const Array& a) : size(a.size), data(a.size != 0 ? new int[a.size] : nullptr)
9  {
10     std::copy(a.data, a.data + size, data);
11 }
12
13 Array(Array && a) noexcept
14 {
15     swap(*this, s);
16     a.data = nullptr;
17 }
```

Débriefing

- 1 Le destructeur n'a pas changé
- 2 Le constructeur de copie n'a pas changé
- 3 On a une fonction `swap`
- 4 L'opérateur d'affectation prend son paramètre par valeur
 - Effectue une copie
 - On utilise `swap` pour échanger `*this` et la copie
- 5 Le constructeur de mouvement utilise `swap` pour échanger `*this` et son paramètre
 - Effet de bord (mais c'est correct)
 - Ne pas oublier (dans ce cas) de mettre
- 6 Pas de besoin nécessaire d'opérateur d'affectation de mouvement
 - On aurait pu en écrire un
 - En l'état, utilise le constructeur de déplacement

L'objectif

- Parfois, un extrait de code a besoin d'acquérir une ressource
 - Un descripteur de fichier
 - Un sémaphore (mutex)
- Il est primordial que la ressource
 - 1 soit bien acquise pour le code qui en a besoin
 - 2 soit bien libérée quand on a fini le traitement
- Ce qui peut mal se passer
 - Erreur d'entrée sortie
 - Impossibilité d'allocation mémoire
 - Exception non rattrapée

L'idiome RAI

Ressource acquisition is initialisation

- Encapsuler une ressource au sein d'une classe
 - Utilisation cette ressource via une instance de la classe
 - Libération de la ressource quand l'objet est détruit
-
- La ressource est souvent initialisée et acquise dans le constructeur
 - La ressource peut être également allouée ailleurs et passée en paramètre
 - L'objet gérant la source est automatique
 - Car ce sont les seuls objets à durée de vie clairement définie au sein d'un bloc
 - Quand l'objet est hors de portée (sortie de bloc), la ressource est libérée
 - Via le destructeur

Exemple académique 1

■ On suppose que m est un mutex

```
1 void wrong()
2 {
3     m.lock();
4     f(); //wrong 1
5     if (!everything_ok())
6         return; //wrong 2
7     m.unlock();
8 }
```

```
1 void right()
2 {
3     std::lock_guard<std::mutex> lk(m); // RAIL class
4     f(); // ok
5     if (!everything_ok())
6         return; // ok
7 }
```

■ Source : Cppreference.com

Exemple académique 2

- On veut implémenter une classe permettant de lire des caractères dans un fichier texte
 - Académique : sans utiliser `std::ifstream`
- Ressources à gérer
 - Le descripteur de fichier
 - Le buffer de lecture
- On va lire avec `cstdio.h`
 - `FILE*` : pointeur vers un descripteur de fichier
 - `fopen` et `fclose` : ouvre et ferme le fichier
 - Obtient le descripteur, ou le libère
 - `fread` : lit des bytes en entrée et les place dans un buffer
- On veut pouvoir déplacer une instance, mais pas la copier
 - Courant comme pratique RAI
 - Limite les problèmes de synchronisation

Exemple académique 2

■ Fichier `rai1.cpp`

```
1  class FileReader {
2      std::FILE* f; //old school (C-style) file management
3      char * buffer;
4      public:
5          FileReader(const char* name) : f(fopen(name, "r+")), buffer(new char[16]) {
6              ...
7          }
8
9          ~FileReader() {
10             std::fclose(f); //flush and free file descriptor
11             if(buffer) {
12                 delete[] buffer;
13                 buffer = nullptr;
14             }
15         }
16
17         FileReader(const FileReader& fh) = delete;
18         FileReader& operator=(const FileReader&) = delete;
19
20         FileReader(FileReader&& fh) noexcept :
21             f(std::move(fh.f)),
22             buffer(std::move(fh.buffer))
23         {
24             fh.buffer = nullptr;
25         }
26     };
```