ISIMA

Références «rvalue» et mouvements

(PARTIE VII – C++11)

Bruno Bachelet

Christophe Duhamel

Rappel sur les *rvalues (1/2)*

- En C++03, deux catégories de valeurs
  - A noter que ces notions évoluent avec les normes du C++

- *lvalue*
  - Historiquement: valeur à gauche (*left-handed*) d'une affectation
  - Valeur «localisable»: accessible via variable, référence ou pointeur
  - Zone mémoire identifiable ⇒ peut être modifiée

- *rvalue*
  - Historiquement: valeur à droite (*right-handed*) d'une affectation
  - Valeur ne pouvant pas être modifiée
  - Typiquement, une valeur à usage ponctuel
    - Littéral: `f(5);`
    - Temporaire construit à la volée: `f(string("Hello !"));`
    - Retour (par copie) d'une fonction: `f(a+b);`

Rappel sur les *rvalues (2/2)*

- En C++03, référence non constante sur une *rvalue* interdite
- Exemple
  - `f(string("Hello !"));`
  - Création à la volée d'un objet ⇒ *rvalue*
- Quel prototype pour récupérer la *rvalue* ?
  - Copie: `void f(string s);` ⇒ OK
  - Référence constante: `void f(const string & s);` ⇒ OK
  - Référence non constante: `void f(string & s);` ⇒ non !

Nouvelle sémantique *(1/2)*

- En C++11, changement de définition
  - Et nouvelles catégories de valeurs: *xvalue*, *prvalue*…
  - `http://en.cppreference.com/w/cpp/language/value_category`
- De manière informelle
  - *lvalue* ⇒ comme avant
  - *rvalue* ⇒ valeur qui peut être modifiée sans effet de bord
    - Cas d'un temporaire
    - Usage unique, donc sa modification est sans conséquence
- Ce qui change concrètement
  - On peut faire une référence non constante sur une *rvalue*
  - On peut volontairement transformer une *lvalue* en *rvalue*

Nouvelle sémantique *(2/2)*

- Nouvelle syntaxe: **`&&`**
  - Référence sur une *rvalue*
  - Il s'agit d'une référence ⇒ mêmes règles que «**`&`**»
    - Caractère constant / non constant
    - Une méthode ne peut pas retourner de référence sur une variable locale

- Retour à l'exemple précédent
  - Rappel: `f(string("Hello !"));`
  - Référence sur *rvalue* non constante: `void f(string && s);` ⇒ OK
  - Mais cette version de la fonction n'est utilisable que pour une *rvalue*

- Pourquoi avoir une référence non constante sur une *rvalue* ?
  - Pour pouvoir la «dépouiller»
  - Autrement dit, récupérer son contenu directement au lieu de le copier
  - La *rvalue* ne sera plus utilisable par la suite

Dépouillement d'objet *(1/3)*

- Exemple

```
class Vecteur {
 private:
  int *    tab_;
  unsigned taille_;

 public:
  explicit Vecteur(unsigned);
  Vecteur(const Vecteur &);
  ~Vecteur(void);

  Vecteur & operator  = (const Vecteur &);
  ...
};
```

Code source complet: `cpp11_move.cpp`

Dépouillement d'objet *(2/3)*

- Dépouiller un vecteur

```
void Vecteur::depouiller(Vecteur && victime) {
 if (tab_) delete [] tab_;
 tab_ = victime.tab_;
 taille_ = victime.taille_;
 victime.tab_ = 0;
 victime.taille_ = 0;
}
```

- Utilisation
  - `Vecteur v1 = ...;`
  - `Vecteur v2 = ...;`
  - `v1.dépouiller(v2)` ⇒ interdit («`v2`» n'est pas une *rvalue*)
  - `Vecteur creerVecteur(void);`
  - `v1.dépouiller(creerVecteur())` ⇒ OK

Dépouillement d'objet *(3/3)*

- Dépouillement ⇒ l'objet n'est plus utilisable…
- …sauf qu'il doit être détruit !

⇒ Conserver une certaine cohérence des objets dépouillés

- Pour que l'appel au destructeur libère bien les ressources restantes

- Autre possibilité: échanger les contenus

```
void Vecteur::depouiller(Vecteur && victime) {
 std::swap(tab_,victime.tab_);
 std::swap(taille_,victime.taille_);
}
```

- Le dépouillement peut s'avérer utile pour optimiser la copie d'objets

Opérateurs de mouvement *(1/3)*

- Exemple: `v3 = v1+v2;`
  - Opérateur «**+**»
    - ⇒ Construction variable locale
    - ⇒ Retour variable locale par copie
  - Affectation
    - ⇒ Copie du retour

- Pire des cas (sans optimisation) ⇒ 2 copies inutiles du tableau
  - Construction par copie + affectation
  - Remarque: l'optimisation devrait éviter la construction par copie du retour

- C++11 introduit 2 nouveaux opérateurs pour optimiser la copie d'objets
  - Constructeur de mouvement / *move constructor*
    - `Vecteur(Vecteur && v)`
  - Affectation de mouvement / *move assignment*
    - `Vecteur & operator = (Vecteur && v)`

Opérateurs de mouvement *(2/3)*

- Constructeur de mouvement

```
Vecteur(Vecteur && v)
: tab_(v.tab_), taille_(v.taille_) {
 v.tab_ = 0;
 v.taille_ = 0;
}
```

- Affectation de mouvement

```
Vecteur & operator = (Vecteur && v) {
 std::swap(tab_,v.tab_);
 std::swap(taille_,v.taille_);
 return *this;
}
```

- Remarque: référence constante sur *rvalue* sans intérêt

Opérateurs de mouvement *(3/3)*

- Sélection automatique de l'opérateur le mieux adapté
  - Pas d'opérateur de mouvement ⇒ opérateur de copie
  - Opérateurs de mouvement + copie disponibles
    - Argument = *lvalue* ou *rvalue* constante ⇒ opérateur de copie
    - Argument = *rvalue* non constante ⇒ opérateur de mouvement
- Quand définir ces opérateurs de mouvement ?
  - Lorsque la copie est coûteuse
  - La STL utilisera ces opérateurs autant que possible
- Sous certaines conditions, opérateurs disponibles par défaut
  - `http://en.cppreference.com/w/cpp/language/move_constructor`
  - `http://en.cppreference.com/w/cpp/language/move_operator`

Conversion en *rvalue (1/2)*

- Comment «forcer» l'utilisation de ces opérateurs ?
  - Possibilité de convertir une *lvalue* en *rvalue* ⇒ **`std::move`**
  - Cela permet de favoriser un mouvement plutôt qu'une copie
  - Mais ensuite l'objet concerné ne doit plus être utilisé

- Exemple

```
template <typename T> inline void swap(T & a,T & b) {
 T tmp = a;
 a = b;
 b = tmp;
}
```

- Trop de copies !
  - Après chaque affectation, la valeur du membre de droite sans intérêt
  - On pourrait donc le dépouiller plutôt que le copier
  - En utilisant les opérateurs de mouvement

Conversion en *rvalue (2/2)*

- Solution potentiellement plus efficace

```
template <typename T> inline void swap(T & a,T & b) {
 T tmp = std::move(a);
 a = std::move(b);
 b = std::move(tmp);
}
```

- Plus aucune copie, mais des mouvements…
  - …à condition que «`T`» implémente les opérateurs de mouvement

- Comment fonctionne «`std::move`» ?
  - Conversion via «`static_cast`»: `T &` → `T &&`
  - Mais il y a quelques subtilités (expliquées plus tard)

- Ne forcer la conversion que si la valeur devient inutile !

Collapsing rules (1/3)

- «**&&**» et les *templates* ⇒ argh !!!
  - **`int &&`** ⇒ *rvalue*
  - **`T &&`** ⇒ *rvalue* ou *lvalue* !

- Nouveauté C++11: les *«collapsing rules»*
  - Règles pour gérer les «références de références»
  - Permettent de ramener les types à leur plus simple expression
  - Voilà les règles utilisées par le compilateur
    - **`T &`** + **`&`** ⇒ **`T &`**
    - **`T &`** + **`&&`** ⇒ **`T &`**
    - **`T &&`** + **`&`** ⇒ **`T &`**
    - **`T &&`** + **`&&`** ⇒ **`T &&`**

- Impact sur la déduction de type à l'instanciation d'un *template*
  - Objectif: faciliter le traitement générique d'arguments
  - Par exemple, pour répondre au problème du *«perfect forwarding»*

Collapsing rules (2/3)

- Comment capter le caractère constant (ou non) d'un argument ?

- `template <typename T> void f(T & x);`
  - N'accepte que les *lvalues* non constantes
  - Alors qu'on cherche à capter les valeurs constantes et non constantes

- `template <typename T> void f(const T & x);`
  - Accepte les *lvalues* et les *rvalues*
  - Mais on force le caractère constant ⇒ perte d'information sur le type

- `template <typename T> void f(T x);`
  - Accepte les *lvalues* et les *rvalues*
  - Mais c'est un passage par copie ⇒ perte d'information sur le type

- Aucune solution idéale en C++03

Collapsing rules (3/3)

- Solution C++11: `template <typename T> void f(T && x);`
- Application des *collapsing rules*
  - `int x;`
  - `const int y;`
  - `f(5)` ⇒ *rvalue* ⇒ `f<int>(int &&)`
  - `f(x)` ⇒ *lvalue* ⇒ `f<int &>(int &)`
  - `f(y)` ⇒ *lvalue* ⇒ `f<const int &>(const int &)`
- Avec les *templates*, «`&&`» ne signifie donc pas toujours *rvalue*
- `T &&` = «référence universelle» (Scott Meyers)
  - Référence *lvalue* ou *rvalue*, constante ou non

Perfect forwarding (1/2)

- Une fois passée en argument, une *rvalue* devient *lvalue*
  - `template <typename T> void f(T && x) { g(x); }`
  - `template <typename T> void g(T &&);`
  - `int x;`
  - `const int y;`

  - `f(5)` ⇒ `g<int &>(int &)` ⇒ caractère *rvalue* perdu
  - `f(x)` ⇒ `g<int &>(int &)`
  - `f(y)` ⇒ `g<const int &>(const int &)`

- *«Perfect forwarding»* ⇒ transfert à l'identique du type
  - Conservation des caractères
    - Constant / non constant
    - *rvalue* / *lvalue*
  - Impossible en C++03

Perfect forwarding (2/2)

- `T &&` ⇒ caractère constant / non constant contenu dans «`T`»
  - Contrairement aux références classiques «`T &`»

- `std::forward` ⇒ conservation du caractère *rvalue* / *lvalue*
  - `template <typename T> void f(T && x)`
    `{ g(std::forward<T>(x)); }`
  - `template <typename T> void g(T &&);`
  - `int x;`
  - `const int y;`

  - `f(5)` ⇒ `g<int>(int &&)`
  - `f(x)` ⇒ `g<int &>(int &)`
  - `f(y)` ⇒ `g<const int &>(const int &)`

Implémentation de «std::move»

- Que se cache-t-il derrière «`std::move`» et «`std::forward`» ?

- Implémentation possible de «`std::move`»
  - ```
    template <typename T> using remove_ref
    = typename std::remove_reference<T>::type;
    ```

  - ```
    template <typename T>
    inline remove_ref<T> && move(T && x) {
     return static_cast<remove_ref<T> &&>(x);
    }
    ```

- `std::remove_reference` $\Rightarrow$ perte du caractère référence du type
  - `remove_ref<T>` = alias pour simplifier l'écriture
  - Retirer la référence est nécessaire à cause des *collapsing rules*
    - `T &` + `&&` $\Rightarrow$ `T &`
    - `remove_ref<T &>` + `&&` $\Rightarrow$ `T &&`

Implémentation de «std::forward»

- Implémentation possible de «`std::forward`»

```
template <typename T>
inline T && forward(remove_ref<T> & x) {
 return static_cast<T &&>(x);
}
```

- *Perfect forwarding* garanti dans un contexte précis
  - «`T`» doit avoir été déduit d'une référence universelle
  - Et seul «`T`» doit servir à l'instanciation du *template*

- Contexte type
  - `template <typename T>`
    `void f(T && x) { g(std::forward<T>(x)); }`
  - Explication
    - `f(5)` ⇒ *rvalue* ⇒ `T = int` ⇒ `static_cast<int &&>`
    - `f(x)` ⇒ *lvalue* ⇒ `T = int &` ⇒ `static_cast<int &>`