Isima

PARTIE VII

Opérations de mouvement

Bruno Bachelet

Loïc Yon

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: `factorielle(5);`
    - Temporaire construit à la volée: `display(string("Hello !"));`
    - Retour (par copie) d'une fonction: `s = add(a,b);`

Rappel sur les *rvalues (2/2)*

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

Nouvelle sémantique *(1/2)*

- Depuis C++11, changement de définition
  - Et nouvelles catégories de valeurs: *glvalue, 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: `display(string("Hello !"));`
  - Référence sur *rvalue* non constante: `void display(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();

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

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_ = nullptr;
 victime.taille_ = 0;
}
```

- Cas d'utilisation
  - `Vecteur v1 = …;`
  - `Vecteur v2 = …;`
  - `Vecteur v3 = …;`
  - `Vecteur add(const Vecteur & a, const Vecteur & b) { Vecteur v = …; return v; }`
  - `v1.depouiller(v2)` ⇒ interdit («`v2`» n'est pas une *rvalue*)
  - `v1.depouiller(add(v2,v3))` ⇒ 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

- Une 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 (retour) + affectation
  - Remarque: l'optimisation évite normalement la construction par copie du retour (cf. *copy elision*, optimisation garantie en C++17 dans certaines conditions)

- Depuis C++11: 2 nouveaux opérateurs pour optimiser la copie
  - 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_ = nullptr;
 v.taille_ = 0;
}
```

- Affectation de mouvement

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

- Remarque: pas d'intérêt à capter spécifiquement une *rvalue* constante

Opérateurs de mouvement *(3/3)*

- Sélection automatique de l'opérateur le plus 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 d'un objet est coûteuse
  - La bibliothèque standard 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 (cf. *collapsing rules*)
- Ne forcer la conversion que si la valeur devient inutile !