ISIMA

PARTIE VIII

Conversion et RTTI

Christophe Duhamel

Bruno Bachelet

Luc Touraille

Implémentation de la conversion

- Deux manières d'implémenter une conversion
- Constructeur avec un seul argument
  - Fournit une conversion implicite
    - `Chaine::Chaine(const char * s);`
    - Conversion implicite `const char *` → `Chaine`
  - Conversion implicite parfois non désirée
    - Mot-clé «`explicit`»
    - `explicit Vecteur::Vecteur(int n);`
- Opérateur de conversion

```
class Chaine {
 ...
 operator char * (void) const { return …; }
 ...
};
```

Politiques de conversion

- Il existe plusieurs opérateurs de conversion
  - `(`*type*`)`
    - Conversion effectuée quoi qu'il arrive
  - `static_cast`
    - Conversion effectuée après vérification à la compilation
  - `dynamic_cast`
    - Conversion effectuée après vérification à l'exécution
  - `const_cast`
    - Conversion portant uniquement sur l'aspect constant
  - `reinterpret_cast`
    - Conversion de pointeurs sans vérification de type

Opérateur *(type) (1/2)*

- Opérateur hérité du C
  - Mais deux syntaxes possibles
  - `c = (Chaine)s;`
  - `c = Chaine(s);`
- Conversion d'objets
  - Effectuée à partir des opérateurs définis par le programmeur
  - Aucun opérateur $\Rightarrow$ conversion interdite
- Conversion de types primitifs
  - Opérateurs de conversion fournis par défaut
- Conversion de pointeurs
  - Toujours autorisée

Opérateur *(type)* (2/2)

- Exemple

```
class A { ... };
class B : public A { ... };
class C { ... };

A * a = new A();
A * b = new B();
C * c = new C();

A * pa; B * pb;
```

- Conversions toujours autorisées
  - `pa = (A *)c;` // (1) Conversion fausse
  - `pb = (B *)a;` // (2) Conversion fausse
  - `pb = (B *)b;` // (3) Conversion ok

- Eviter l'utilisation de l'opérateur `(`*type*`)`
  - Cas (1): détection possible à la compilation
    - Utiliser l'opérateur `static_cast`
  - Cas (2) & (3): détection à l'exécution
    - Utiliser l'opérateur `dynamic_cast`

Opérateur *static_cast*

- Vérifie la conversion de pointeurs (ou de références) à la compilation
- Conversion autorisée s'il y a un lien d'héritage
  - `pb = static_cast<B *>(a);` // Autorisé
  - Même si cela risque d'être invalide à l'exécution
  - Conseil: utiliser `dynamic_cast` (pour une vérification à l'exécution)
- Conversion refusée s'il n'y a pas de lien d'héritage
  - `pa = static_cast<A *>(c);` // Refusé
  - ```
    int * pi = …;
    float * pf = static_cast<float *>(pi);
    ```
    // Refusé
- Fonctionne de la même manière sur les références
- Conversion vers `void *` autorisée
  - `void * pv = static_cast<void *>(a);`
- Conversion depuis `void *` devrait être refusée
  - `pa=static_cast<A *>(pv);`
  - Peut être autorisé suivant le compilateur
  - Conseil: utiliser `reinterpret_cast` dans cette situation
    ```

Opérateur *dynamic_cast*

- Vérification de la conversion de pointeurs (ou de références) à l'exécution
  - La même vérification que `static_cast` est effectuée à la compilation
  - Il ne peut pas être employé sur le type `void *`
- Utilisé lors d'une conversion descendante (*downcast*)
  - Conversion d'une classe mère vers une classe fille
  - Conversion ascendante (fille→mère) toujours possible
- A l'exécution, la conversion peut échouer
  - Conversion de pointeurs ⇒ pointeur nul retourné
  - Conversion de références ⇒ exception levée
  - `pb = dynamic_cast<B *>(a);`
- Exemple avec les références

```
A a;
B b;

A & ra = a;
A & rb = b;
B & ref1 = dynamic_cast<B &>(ra); // Exception levée à l'exécution
B & ref2 = dynamic_cast<B &>(rb); // Conversion ok
```

- Conversion par référence évite les recopies

Opérateur *const_cast*

- Permet de retirer l'aspect constant d'un objet

- N'a pas de signification sur une variable objet
  - ```
    const Chaine c1;
    Chaine c2 = c1;
    ```
  - La conversion ne pose aucun problème
  - Car une copie est effectuée,
    et elle ne possède pas l'aspect constant

- Vraiment utile pour les références
  - ```
    const Chaine c1;
    Chaine & c2 = const_cast<Chaine &>(c1);
    ```
  - `const_cast` indispensable ici pour autoriser la conversion

- L'usage de cet opérateur est à éviter
  - Il permet de briser des règles fondamentales
  - Souvent, obligation d'utiliser `const_cast` $\Rightarrow$ erreur de conception
    - Soit en imposant à tort la constance sur la variable
    - Soit en omettant des méthodes qui permettraient un accès non constant
  - La solution à votre problème est peut-être le modificateur `mutable`

Conversions: conclusion

	`Chaine` vers `char *`	`B *` vers `A *`	`A *` vers `B *`	`Objet *` vers `void *`	`void *` Vers `Objet *`
`(type)`	Oui	Oui	Oui	Oui	Oui
`static_cast`	Oui	Oui	Oui	Oui	Ne devrait pas
`dynamic_cast`	Non applicable	Oui	Oui (après vérification)	Oui	Non applicable
`reinterpret_cast`	Non applicable	Oui	Oui	Oui	Oui

Mécanisme RTTI *(1/3)*

- *Run-Time Type Information*
- Très utile pour déterminer la classe réelle d'un objet à l'exécution
  - Celui-ci doit être pointé ou référencé
- Même type de contrôle que **`dynamic_cast`**
- Mot-clé **`typeid`** retourne une structure de type **`type_info`**
  - **`#include <typeinfo>`**
- Exemple

```
Poisson p("Maurice",10,20,3);
Mammifere m("Rantanplan",5,9,17);
Animal * pa = &p;
Animal * pb = &m;
...
std::cout << typeid(*pa).name();
```

Mécanisme RTTI *(2/3)*

- La structure **`type_info`** contient des informations sur le type
  - Nom du type: méthode **`name`**
  - Plus intéressant, opérateurs **`==`** et **`!=`**

- Permet de vérifier que deux objets sont du même type

```
if (typeid(*pa)==typeid(*pb))
 cout << "Ils sont de même type." << std::endl;
else
 cout << "Ils ne sont pas de même type." << std::endl;
```

- **`typeid`** peut s'appliquer sur un type

```
if (typeid(*pa)==typeid(Poisson))
 std::cout << "C'est un poisson.";
else std::cout << "Ce n'est pas un poisson.";
```

- Attention au piège: fournir des références et non des pointeurs
  - Car pas de liens entre les pointeurs
  - Aucun lien entre **`Animal *`** et **`Poisson *`**

Mécanisme RTTI *(3/3)*

- Exemple

```
Animal * pp = new Poisson("Maurice",10,20,3);
Animal & rp = *pp;
```

- Résultats de comparaisons de types

	typeid(Animal)	typeid(Poisson)	typeid(Animal *)	typeid(Poisson *)
typeid(pp)	!=	!=	==	!=
typeid(rp)	!=	==	!=	!=
typeid(*pp)	!=	==	!=	!=
typeid(&rp)	!=	!=	==	!=