



## Apprendre C++ avec Qt : Annexe 5 Run-Time Type Information

Le RTTI est un dispositif qui permet de déterminer le type de l'objet dont l'adresse est contenue dans un pointeur sur une classe de base (il peut légitimement s'agir d'une instance de la classe de base ou d'une instance de n'importe laquelle des classes dérivées de celle-ci).

Etant donné qu'un pointeur sur une classe de base ne doit normalement servir qu'à appeler des fonctions de cette classe (qui peuvent, éventuellement, être virtuelles et redéfinies dans les classes dérivées), il n'est habituellement pas nécessaire de connaître le type exact de l'objet qu'il désigne (la virtualité des fonctions redéfinies suffit à garantir que c'est le code adapté au type concerné qui sera exécuté). Nous pouvons donc conclure que

Un programme "normal" ne fait jamais appel au RTTI.

S'il s'agit de déboguer un programme ou simplement d'explorer le fonctionnement des fonctions virtuelles, le recours au RTTI peut cependant s'avérer instructif, ou même très utile.

### 1 - L'opérateur typeid et la classe type\_info

L'opérateur typeid() renvoie une référence à une constante de type type\_info correspondant au type de l'objet auquel il est appliqué.

Typiquement, cet objet est désigné en déréférençant un pointeur sur une classe de base.

Une valeur de type type\_info décrit un type sous une forme utilisable par un programme. Il est ainsi possible de comparer deux valeurs (avec les opérateurs == et !=) et d'obtenir une chaîne de caractères correspondant au nom utilisé dans le code source pour désigner le type. Ainsi, après

```
1 class CBase
2 {
3 public:
4     virtual void m_f() {}
5 };
6 class CDerivee : public CBase
7 {
8 public:
9     int m_propre;
10};
```

l'exécution du fragment de code suivant

```
1 CDerivee unD;
2 CBase * ptr = &unD;
3 const type_info & leType = typeid(*ptr);
4 QString laClasse = leType.name();
```

placera dans la variable laClasse la chaîne class CDerivee

L'utilisation de la classe type\_info exige la présence d'une directive #include "typeinfo.h"

Si vous utilisez Visual C++ 6.0, vous devez aussi cocher la case "Enable RTTI" dans l'onglet "C/C++" (category "C++ language") de vos "Projects settings".

Notez également que

L'opérateur typeid() ne donne réellement le type de l'objet pointé que si la classe de base concernée possède au moins une fonction virtuelle.

## 2 - L'opérateur `dynamic_cast` < >

Lorsqu'un pointeur sur une classe de base contient l'adresse d'une instance d'une classe dérivée, l'opérateur `dynamic_cast` < > permet de convertir cette valeur en un pointeur sur la classe dérivée en question (ce qui permettra d'accéder aux membres propres qu'elle définit). Si l'objet pointé n'est pas du type attendu (il peut s'agir d'une simple instance de la classe de base, par exemple), la valeur produite par `dynamic_cast` < > est nulle.

C'est cette dernière caractéristique qui justifie l'existence de `dynamic_cast` < >. L'usage d'un `static_cast` < > ou d'un `reinterpret_cast` < > permet en effet d'obtenir n'importe quelle conversion entre type de pointeurs, mais sans aucune vérification de cohérence...

```

1 //les classes CBase et CDerivee sont supposées définies comme précédemment
2 void fonctionSauvage(CBase *ptrBase)
3 {
4     CDerivee * ptrDerivee = dynamic_cast<CDerivee *> (ptrBase);
5     if (ptrDerivee != NULL)
6         ptrDerivee->m_propre = 4;
7 }
```

Si la fonction ci-dessus permet d'illustrer l'usage de l'opérateur `dynamic_cast` < >, elle permet également d'expliquer pourquoi ce type de conversion (connue sous le nom de "downcasting", parce qu'elle permet de "descendre" dans la hiérarchie des classes) n'est généralement pas souhaitable.

Remarquons tout d'abord que cette fonction, qui n'est membre ni de `CBase` ni de `CDerivee`, prétend accéder à une variable membre. Ce simple fait est déjà une hérésie, puisque l'interface d'une classe ne doit comporter que des fonctions membre.

En quoi la situation serait-elle différente s'il s'agissait d'appeler une fonction membre de `CDerivee` et non d'accéder à une de ses variables ? Dans ce contexte, la simple évocation d'une fonction membre appelle la question "Pourquoi n'est-elle pas virtuelle ?"

Si la classe `CBase` comporte une fonction virtuelle nommée `remiseEnEtatInitial()`, par exemple, cette fonction peut être redéfinie par les classes dérivées, qui sont les seules à vraiment savoir ce qu'il convient de faire avec leurs membres propres.

La version redéfinie dans `CDerivee` se chargera, par exemple, d'affecter la valeur 4 à la variable `m_propre`.

Le code de la fonction `fonctionSauvage()` devient alors

```

1 void fonctionSauvage(CBase *ptrBase)
2 {
3     ptrBase->remiseEnEtatInitial();
4 }
```

Non seulement ce code est plus court et plus clair, mais il fait quelque chose dont la version initiale de la fonction était bien incapable : il traite correctement le cas de *toutes* les classes dérivées de `CBase`, qu'elles soient présentes ou à venir.

Il est concevable que la fonction propre qui doit être exécutée soit tellement spécifique à la classe dérivée concernée que la création d'une fonction virtuelle dans la classe de base semble franchement déraisonnable. Une autre piste doit alors être explorée : pourquoi la fonction `fonctionSauvage()` n'est-elle pas elle-même une fonction virtuelle de la classe de base ? Elle pourrait alors être facilement redéfinie par les classes nécessitant des traitements très spécifiques (et par elles seules), et ces redéfinitions pourraient, le cas échéant, appeler explicitement la version de base de la fonction, pour éviter de s'occuper d'autre chose que de la spécificité qui justifie leur existence.

Nous pouvons donc conclure que

Un programme "normal" ne fait jamais appel à l'opérateur `dynamic_cast` < >.