

Boost.Conversion : les conversions non numériques de Boost

par Matthieu Brucher (<http://matthieu-brucher.developpez.com/>) (Blog)

Date de publication : 19/06/2006

Dernière mise à jour : 31/08/2006

Les opérateurs de conversion présentés ici permettent de caster des objets dans une hiérarchie avec des avantages sur les casts classiques du C++, ou de transformer un nombre en string et inversement.

Introduction

I - Les conversions polymorphiques

I-A - polymorphic_cast<>

I-B - polymorphic_downcast<>

II - Les conversions lexicales

Conclusion

Introduction

Boost.Conversion propose quelques opérateurs de conversion qui permettent de simplifier les conversions polymorphiques ou lexicales classiques.

Cette bibliothèque est templatée et n'utilise pas de bibliothèque compilée .lib/.a ou .dll/.so. Toutes les classes et les fonctions vivent dans le namespace **boost**.

I - Les conversions polymorphiques

Deux opérateurs de conversions polymorphiques, avec des caractéristiques bien distinctes. Les opérateurs classiques - des vrais opérateurs car les convertisseurs de Boost ne sont que des fonctions - ont des fonctionnalités bien définies eux aussi. **static_cast<>** fait une conversion sans vérifier si c'est possible ou pas tandis que **dynamic_cast<>** retourne un pointeur nul lorsqu'il est utilisé sur des pointeurs, ce qui n'est pas forcément le plus utile.

I-A - polymorphic_cast<>

Cet opérateur est à utiliser uniquement sur les pointeurs. En fait, il s'agit d'un **dynamic_cast<>** déguisé qui lèvera une exception de type **std::bad_cast** si le pointeur retourné par **dynamic_cast<>** est nul. Il faut donc bien se rendre compte que le comportement normal n'est pas de lever une exception. Si on s'attend à ce que la conversion échoue, il faut conserver **dynamic_cast<>**.

I-B - polymorphic_downcast<>

Cet opérateur propose une fonction différente en mode release - avec définition de **NDEBUG** - ou en mode debug. En mode release, **polymorphic_downcast<>** retourne un **static_cast<>** sur le pointeur en entrée. En revanche, en mode debug, une assertion à l'aide de **dynamic_cast<>** permet de vérifier que le pointeur qui sera retourné est bien valide. Il n'y a donc une vérification qu'en mode debug.

II - Les conversions lexicales

Ces convertisseurs permettent de transformer un nombre en chaîne de caractères et inversement. L'appel se fait simplement - exemples de la doc de Boost - :

cast de char* vers des short

```
int main(int argc, char * argv[])
{
    using boost::lexical_cast;
    using boost::bad_lexical_cast;

    std::vector<short> args;

    while(*++argv)
    {
        try
        {
            args.push_back(lexical_cast<short>(*argv));
        }
        catch(bad_lexical_cast &)
        {
            args.push_back(0);
        }
    }
}
```

cast de int vers string

```
void log_message(const std::string &);

void log_errno(int yoko)
{
    log_message("Error " + boost::lexical_cast<std::string>(yoko) + ": " + strerror(yoko));
}
```

On voit dans ces exemples qu'il y a un mécanisme d'erreur lorsque la conversion est impossible, puisque **lexical_cast<>** lève une exception de type **bad_lexical_cast<>**, type dérivé de **bad_cast<>**. La conversion utilise des **stringstream** en interne, la méthode classique utilisée dans ces cas, sauf que tout est encapsulé avec une vérification d'erreur.

Conclusion

Autrefois, il y avait un **numeric_cast**<> pour transformer un type de nombre en un autre avec levée d'exception, mais cet opérateur est dans une autre bibliothèque. Cela rend Boost.Conversion plus intrinsèque et limite sa responsabilité aux conversions polymorphiques, où l'ajout des convertisseurs amène la sécurité par levée d'exception ou une rapidité au prix d'une vérification en mode debug, ainsi qu'aux conversions lexicographiques où à défaut d'être la plus performante, la bibliothèque permet une gestion des erreurs plus sûre que les **printf**, **scanf**, **strol**, ... hérités du C qui ne peuvent lever une exception.

