

# Boost.StaticAssert : les assertions statiques de Boost

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

Date de publication : 03/07/2006

Dernière mise à jour : 31/08/2006

Les assertions statiques sont des assertions exécutées à la compilation. Elles permettent de vérifier que les types de données utilisés sont bien valables, donc très utiles pour les utilisations des classes templates.

## Introduction

I - Utilisation dans un espace de noms

II - Utilisation dans une fonction

III - Utilisation dans une classe

IV - Utilisation dans un template

Conclusion

## Introduction

L'en-tête `<boost/static_assert.hpp>` fournit une simple macro **BOOST\_STATIC\_ASSERT(x)** qui génère un message d'erreur si l'expression entière constante **x** ne vaut pas **true**. C'est donc l'équivalent à la compilation de la macro **assert**. Lors de la compilation, si la génération est effectuée, la classe est vide et donc ne prend pas de place mémoire. De même, la macro n'est évaluée que lors de l'instanciation, donc pas de problème avec les classes templates.

On va regarder son utilisation à 4 emplacements, dans un espace de noms, dans une fonction, dans une classe et dans des templates.

## I - Utilisation dans un espace de noms

Ici, la macro sera toujours instanciée. Cela signifie que les assertions dans les espaces de noms servent à vérifier les impératifs de la plateforme - taille d'un `int`, version d'une bibliothèque, ... -. Comme la macro **BOOST\_STATIC\_ASSERT(x)** est en fait un **typedef** avec l'utilisation de la macro `_LINE_`, l'utilisation dans un espace de noms peut entraîner la duplication des **typedefs** et dans certains cas des warnings, donc utilisez un espace de nom particulier pour tester ces conditions, et dans un seul fichier d'en-tête.

```
#include <boost/static_assert.hpp>

namespace conditions
{
    BOOST_STATIC_ASSERT(sizeof(char)==1);
}
```

## II - Utilisation dans une fonction

Dans le cadre des fonctions, on pense principalement aux fonctions templates. Cela permet, logiquement, de vérifier le type des données passées en paramètres, comme la vérification de capacités telles la conversion vers un type dont on connaît les caractéristiques - itérateurs à accès aléatoire -, la taille, ...

```
#include <boost/static_assert.hpp>

template<typename T>
T test(const &T)
{
    BOOST_STATIC_ASSERT(sizeof(T)>1);
    return T;
}
```

### III - Utilisation dans une classe

L'utilisation dans une classe a les mêmes capacités que l'utilisation dans le cadre d'une fonction.

```
#include <boost/static_assert.hpp>

template<typename T>
class test
{
    BOOST_STATIC_ASSERT(sizeof(T)>1);

public:
    test(const &T)
    {
        // ...
    }
}
```

## IV - Utilisation dans un template

Lors de l'utilisation dans un template, normalement l'erreur ne survient qu'à l'instanciation. Malheureusement, certains compilateurs ne fonctionnent pas ainsi. Par exemple, ce code :

```
template <class T>
struct must_not_be_instantiated
{
    BOOST_STATIC_ASSERT(false);
};
```

Produit une erreur et on est obligé de le remplacer par :

```
template <class T>
struct must_not_be_instantiated
{
    BOOST_STATIC_ASSERT(sizeof(T) == 0);
};
```

On voit ici qu'une telle assertion peut aussi être utilisée pour empêcher l'utilisation d'une classe template non spécialisée.

## Conclusion

La simplicité et la légèreté de cette macro fait qu'on devrait l'utiliser partout où une assertion effectuée par **assert** pourrait être faite à la compilation. Même si c'est rare, on gagne du temps car l'assertion est à la compilation, non pas à l'exécution. De même, l'assertion est plus générale car elle vaut sur toutes les possibilités des données en entrée et non pas dans un cas d'utilisation comme c'est le cas des assertions à l'exécution. Enfin, c'est la seule manière de vérifier qu'on instancie bien le bon type de données.

