

# Boost.Array : Tableau de taille fixe STL-like

par Alp Mestan ([Site perso de Alp](#)) ([Blog](#))

Date de publication :

Dernière mise à jour :

Cet article a pour but de présenter une classe fournie avec les bibliothèques Boost : Boost.Array, afin que vous sachiez bien la manipuler si l'occasion se présente dans l'un de vos codes.

I - Introduction

II - Référence

II-A - Interface complète

II-B - Définition de types

II-C - Taille et Opérateur d'affectation

II-D - Itérateurs

II-E - Accès aux données

II-F - Modification

II-G - Opérateurs

II-H - Mise en oeuvre de boost::array

III - Exemples

IV - Conclusion

V - Remerciements

## I - Introduction

Il arrive quelques fois d'avoir besoin d'un tableau de taille fixe ou au pire dont la taille ne dépassera pas une certaine limite à coup sûr. Et dans ces cas là, un certain nombre de personnes utilisent la classe standard `std::vector` en appelant dès sa création la fonction membre `std::vector<T>::resize` afin de ne plus avoir à réallouer de mémoire. Cependant, cette classe n'est pas faite pour ça, au sens où elle n'est pas optimisée pour ce genre d'utilisation. Les tableaux bruts, du genre `T my_array[N]`, suffisent mais ne présentent aucune interface, n'étant pas encapsulés dans une classe, ce qui rend leur utilisation moins agréable et cela rajoute des lignes de code. L'utilisation mentionnée ci-dessus est parfaitement adaptée à `boost::array`, qui encapsule un tableau brut et se comporte comme tel grâce à la surcharge d'opérateurs. De plus on dispose d'opérateurs pour tester l'égalité de deux tableaux, par exemple, ainsi que de fonctions membres pour assigner une valeur à tous les éléments du tableau, pour échanger deux `boost::array`... Et enfin, la classe `boost::array` comporte une interface semblable à celle des conteneurs de la STL utilisant le concept d'itérateur pour le parcours du tableau. On peut donc très facilement utiliser tous les algorithmes standards et autres algorithmes basés sur le même modèle sur un `boost::array`, ce qui facilite grandement les traitements sur les éléments qu'il contient, en raccourcissant le code que l'on écrira pour cela. Nous allons maintenant voir comment se présente l'interface de cette classe puis nous verrons quelques exemples d'utilisation de cette dernière.

## II - Référence

### II-A - Interface complète

Nous allons premièrement voir l'allure de la classe modèle (*template*) `boost::array`. Voici donc l'interface de la classe `boost::array`.

#### Interface de `boost::array`

```
template <typename T, std::size_t N>
class array {
public:
    // types
    typedef T value_type;
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef T& reference;
    typedef const T& const_reference;
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;

    // static constants
    static const size_type static_size = N;

    // construct/copy/destruct
    template<typename U> array& operator=(const array<U, N>&);

    // iterator support
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    // reverse iterator support
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;

    // capacity
    size_type size();
    bool empty();
    size_type max_size();

    // element access
    reference operator[](size_type);
    const_reference operator[](size_type) const;
    reference at(size_type);
    const_reference at(size_type) const;
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
    const T* data() const;
    T* c_array();

    // modifiers
    void swap(array<T, N>&);
    void assign(const T&);

    T elems[N];
};
```

## Interface de boost::array

```
// specialized algorithms
template<typename T, std::size_t N> void swap(array<T, N>&, array<T, N>&);

// comparisons
template<typename T, std::size_t N>
    bool operator==(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
    bool operator!=(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
    bool operator<(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
    bool operator>(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
    bool operator<=(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
    bool operator>=(const array<T, N>&, const array<T, N>&);
```

### Paramétrisation de boost::array par :

- 1 le type des objets qu'elle stockera, ici *T*;
- 2 la taille du tableau, car boost::array est un tableau de taille fixe, représentée ici par *N*.

Nous allons maintenant analyser l'interface de cette classe morceau par morceau.

## II-B - Définition de types

Dans l'interface de boost::array, on voit donc un bloc de définitions de types, qui est le suivant.

### Définitions de types dans boost::array

```
// types
typedef T value_type;
typedef T* iterator;
typedef const T* const_iterator;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
typedef T& reference;
typedef const T& const_reference;
typedef std::size_t size_type;
typedef std::ptrdiff_t difference_type;
```

*T* est évidemment le type contenu dans le tableau. Ensuite, on voit que le type *boost::array<,N>::iterator* est en fait un simple pointeur vers *T*, car pour itérer sur un tableau, il suffit de pointer sur un élément du tableau, et des opérateurs d'incrémentations par exemple pour se déplacer sur les cases du tableau. De même, on crée le type *const\_iterator* pour se déplacer sur les cases du tableau en assurant au compilateur qu'on ne modifiera pas les objets contenus dans le tableau. On a également deux types correspondants pour le déplacement dans le sens inverse, de la fin vers le début, ainsi que des types nous permettant de prendre des références sur les éléments du tableau. Pour terminer, nous avons les types *size\_type* et *difference\_type* qu'on utilisera respectivement pour les index du tableau et sa taille ainsi que pour utiliser l'arithmétique des pointeurs sur le tableau.

## II-C - Taille et Opérateur d'affectation

Après ce bloc de définitions de types, on voit un membre statique de type `size_t` donnant accès à la taille du tableau statique, qui est le deuxième paramètre de la classe `boost::array`. Plus intéressant, on voit une déclaration d'opérateur d'affectation, paramétrisé par un type `U`, permettant d'assigner au `boost::array` courant un `boost::array` de même taille mais contenant un type différent. Il est cependant important de savoir que cet opérateur n'est utilisable que s'il existe une conversion de `U` vers `T`.

## II-D - Itérateurs

Ensuite, nous constatons qu'il y a un autre bloc de déclarations concernant les itérateurs, qui je pense peut se passer de commentaires. Ce bloc est important pour rendre `boost::array` compatible avec les algorithmes de la bibliothèque standard et autres algorithmes suivant le même modèle.

## II-E - Accès aux données

Nous voyons maintenant un bloc concernant l'accès aux données par référence, en opposition au bloc se trouvant au-dessus permettant l'accès à travers des itérateurs. On note notamment la définition de l'opérateur `[]`, permettant une utilisation transparente de classe alors qu'il peut nous sembler utiliser un tableau brut. Nous voyons également que l'on peut accéder au tableau par un `const T*`, nous permettant d'accéder donc en lecture seule au tableau, grâce à la fonction `data()`. La fonction `c_array()` retourne un pointeur sur le premier élément permettant un accès en lecture/écriture sur le tableau.

## II-F - Modification

Nous disposons de deux fonctions membres permettant la modification globale du tableau. La première, `swap`, nous permet d'échanger deux `boost::array` de même taille et contenant le même type d'éléments. La deuxième, `assign`, nous permet d'assigner la même valeur à tous les éléments de notre tableau. Enfin, nous disposons d'une fonction hors de la classe permettant d'échanger deux `boost::array`, qui est utilisée par la fonction membre `swap`.

## II-G - Opérateurs

Nous avons également à notre disposition un certain nombre d'opérateurs, dont le rôle est de faciliter l'utilisation de notre classe. Par exemple, pour tester l'égalité de deux `boost::array`, nous avons simplement à écrire le code suivant.


### Egalité de deux `boost::array`

```
boost::array<int,15> tab1;
boost::array<int,15> tab2;
// on remplit chacun des tableaux
// et on compare
if(tab1==tab2)
{
    std::cout << "Les deux tableaux sont identiques" << std::endl;
}
```

## II-H - Mise en oeuvre de `boost::array`

Maintenant que nous avons globalement expliqué le fonctionnement de la classe `boost::array` et avons pris connaissance de son interface, il est temps de la mettre en oeuvre dans des exemples, dans la deuxième partie de cet article.

### III - Exemples

 Avant toute chose, je tiens à rappeler que le paramètre donnant la taille du tableau, noté *N* ci-dessus, doit être donné à la compilation.

Pour faciliter votre compréhension, quelques exemples d'utilisations de `boost::array` sont donnés ci-dessous. N'hésitez surtout pas à les tester, à modifier le code et recompiler pour vérifier que vous avez bien compris chaque code.

#### Remplir un `boost::array` et afficher son contenu

```
#include <iostream>
#include <boost/array.hpp>
#include <iterator>
#include <algorithm>

using namespace std;

int main()
{
    boost::array<int,10> my_array;
    my_array.assign(42);
    copy(my_array.begin(),my_array.end(),ostream_iterator<int>(cout," "));
    return 0;
}
/* On remplit le tableau avec la valeur 42, puis on l'affiche */
```

#### Utilisation de l'opérateur []

```
#include <iostream>
#include <boost/array.hpp>
#include <iterator>
#include <algorithm>

using namespace std;

int main()
{
    boost::array<int,10> my_array;
    my_array.assign(42);
    for(int i = 0; i < 10; i++)
    {
        if(!(i%2))
        {
            my_array[i] = 24;
        }
    }
    copy(my_array.begin(),my_array.end(),ostream_iterator<int>(cout," "));
    return 0;
}
/* On remplit le tableau avec la valeur 42, puis on remplace une fois sur deux
 * par la valeur 24. Pour terminer on affiche.
 * /
```

#### Utilisation des algorithmes standards

### Utilisation des algorithmes standards

```
#include <iostream>
#include <boost/array.hpp>
#include <iterator>
#include <algorithm>

using namespace std;

struct Init
{
    int i;
    Init() : i(0) { }

    int operator() (int)
    {
        return i++;
    }
};

struct TwoTimes
{
    int operator() (int i)
    {
        return i*2;
    }
};

int main()
{
    boost::array<int,10> my_array;

    transform(my_array.begin(),
              my_array.end(),
              my_array.begin(),
              Init());

    copy(my_array.begin(),my_array.end(),ostream_iterator<int>(cout," "));

    cout << endl;

    transform(my_array.begin(),
              my_array.end(),
              my_array.begin(),
              TwoTimes());

    copy(my_array.begin(),my_array.end(),ostream_iterator<int>(cout," "));

    return 0;
}

/* Ce programme initialise le tableau avec des valeurs consécutives en partant de 0
 * puis transforme le contenu en remplaçant chaque valeur par son double
 * /
```



## IV - Conclusion

Maintenant que vous savez vous servir de *boost::array*, il faut apporter quelques précisions quant à son utilisation. Cette classe est, ne l'oubliez pas, une enveloppe pour un tableau de taille fixe. Cette classe n'est pas à utiliser comme un *std::vector*, qui lui est fait pour qu'on lui rajoute ou qu'on lui enlève des éléments. Les avantages de *boost::array* sont, lorsqu'on l'utilise de manière adaptée à ce pour quoi il a été conçu, qu'il ne réalloue pas de mémoire, contrairement à *std::vector*, tout en présentant la même facilité d'utilisation et en étant compatible à l'utilisation des algorithmes standards. Son désavantage est justement qu'on ne peut pas rajouter ou enlever des éléments, du fait qu'on travaille avec un tableau de taille fixe, qui alloue une bonne fois pour toute un bloc en mémoire à l'instanciation d'un *boost::array*. Il faut donc savoir utiliser la bonne classe au bon moment, car *std::vector* et *boost::array* ne possèdent pas du tout le même champs d'application.

| Classe                               | Utilisation   |
|--------------------------------------|---|
| <code>std::vector&lt;T&gt;</code>    | Tableau dynamique - Ajout/Suppression d'éléments à la volée   |
| <code>boost::array&lt;T,N&gt;</code> | Tableau statique de N éléments - Action sur un nombre fini d'éléments, sans en enlever ou en rajouter |

## V - Remerciements

Merci à **Laurent**, **khayyam90** et **Corbase** pour m'avoir aidé à améliorer cet article.

