

# Présentation de boost::assign

par Come David ([Site perso de David Come](#))

Date de publication : 12/02/2007

Cet article a pour but de vous présenter boost::assign

I - Introduction

II - Référence

    II-1 - Interface de la classe

    II-2 - Détails des fonctions

III - Exemples

IV - Conclusion

## I - Introduction

Un des grands avantages du C++ sur le C est la présence des classes de la *STL* telles que *std::vector* ou *std::list* qui nous facilitent la vie à long terme. Or un des grands reproches que l'on peut leur faire est la perte d'une initialisation *C-like*, c'est-à-dire de la forme :

```
char const* k_messages[] = { "trop court", "trop long", "super"};
```

Essayez de faire cela avec un *std::vector* de *std::string* et regardez la tête de votre compilateur. Il n'aime pas trop. Mais la solution à ce problème passe par *boost::assign*, petite partie de *boost* permettant de retrouver une syntaxe assez proche de celle ci. Mais ce n'est pas son seul rôle. En effet, elle permet plein d'autres choses dans le domaine de l'affectation de valeurs aux conteneurs.

## II - Référence

### II-1 - Interface de la classe


Avant toute chose, regardons ce que propose `boost::assign` : Pour les conteneurs de la STL:

- `operator+=`
- `operator()`
- `list_of()`
- `map_list_of()`
- `repeat()`, `repeat_fun()` and `range()`
- `ref_list_of()` and `cref_list_of()`

Pour `Boost::Pointer_Container` :

- `ptr_push_back()`, `ptr_push_front()`, `ptr_insert()` and `ptr_map_insert()`
- `ptr_list_of()`

Comme vous pouvez le voir, les fonctions proposées se décomposent en 2 grandes catégories : Une pour les conteneurs de la STL (`std::list`, `std::vector`, `std::map`, et leurs adaptateurs) et une 2eme pour `boost::Pointer_Container`.

 *boost::Pointer\_Container est une classe de boost offrant des capacités similaires à un `std::vector` sauf qu'elle est spécialisé dans le stockage de pointeurs.*

### II-2 - Détails des fonctions

**operator+=** Il permet de remplir un conteneur de façon intuitive en faisant `conteneur+=valeur0,valeur1,valeurn` ;

**operator()** Inversement à `operator+=()` que l'on appelait directement sur le conteneur, on va passer par un objet proxy pour utiliser `operator()`. La fonction pouvant utiliser `operator()` sur un conteneur tire son nom d'après la fonction membre d'insertion dans le conteneur. Donc pour utiliser `operator()` sur un vecteur nommé `vec`, on fera `push_back(vec)(objet0)(objet1)(objetN)`; alors que sur une map, on utilisera `insert (conteneur) (objet)`;

 *Par défaut, jusqu'à 5 arguments sont supportés par `operator()` mais il est possible de changer cela en définissant la macro `BOOST_ASSIGN_MAX_PARAMS`*

**list\_of**: Contrairement à ce que l'on a vu avant, `list_of` ne sert pas à remplir un conteneur mais bien à l'initialiser, c'est à dire à lui donner des valeurs de départ. Avec `list_of`, on crée une liste anonyme automatiquement convertie pour n'importe quel conteneur séquentiel (`vector`, `list`, ...) . Dans le cas où on doit initialiser un adaptateur de conteneur, il suffit d'appeler après le dernier objet la fonction `to_adapter()`.

**map\_list\_of** : La même chose que `list_of` mais pour les map.

**tuple\_list\_of**: Dans le même esprit que `list_of` et `map_list_of` mais pour `boost::tuple` cette fois.

**repeat(), repeat\_fun() and range()**. Ces fonctions ne s'utilisent pas toutes seules, elles sont plutôt des utilitaires (principalement pour `operator+=` ). `repeat(X,val)` permet de répéter X fois la valeur tandis que `repeat_fun(X,func)`

permet de répéter X fois func. Enfin range permet d'insérer un intervalle d'itérateur voire un conteneur en entier (dans ce cas l'intervalle ira de begin() à end() )

**ref\_list\_of** permet de créer très rapidement un intervalle anonyme à partir de différentes variables. Elle renvoie cet intervalle.

**cref\_list\_of()** fait la même chose que ref\_list\_of() , l'attribut const en plus.

**ptr\_push\_back, ptr\_push\_front, ptr\_insert and ptr\_map\_insert** ont le même rôle que leurs homologues, sauf qu'elles travaillent avec des pointeurs et ont une résistance aux exceptions.

**ptr\_list\_of:** La seule différence avec son homologue est en cas d'utilisation avec un adaptateur de conteneur, là il faut passer le conteneur en paramètre à to\_adapter.

Et c'est tout ! Voyons tout de suite ce que peut donner cette bibliothèque.

### III - Exemples

Voici quelques exemples d'utilisation : Concaténer des conteneurs :

```
#include <vector>
#include <iostream>
#include <boost/assign.hpp>
#include <boost/array.hpp>

using namespace boost::assign;
using namespace std;

int main()
{
    boost::array<int,5> array1;  array1.assign(42);
    boost::array<int,5> array2;  array2.assign(51);

    std::vector<int> result;

    push_back(result).range(array1).range(array2);

    cout<<"s: " <<result.size()<<endl;

    for(int i=0;i<10;i++)
        cout<<result[i]<<" " <<i<<endl;

    return 0;
}
```

En comparaison, voici le code utilisant les algorithmes standards

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

using namespace std;

int main()
{
    boost::array<int,5> array1;  array1.assign(42);
    boost::array<int,5> array2;  array2.assign(51);

    std::vector<int> result;
    #if METHODE
    result.resize(array1.size() + array2.size());

    copy(array1.begin(),array1.end(),result.begin());
    copy(array2.begin(),array2.end(),result.begin());


    #else
    copy(array1.begin(),array1.end(),std::back_inserter< std::vector<int> >( result ));
    copy(array2.begin(),array2.end(),std::back_inserter< std::vector<int> >( result ));
    #endif

    const int size=result.size();
    cout<<"size: " <<size<<endl;

    for(int i=0;i<size;i++)
        cout<<result[i]<<endl;
}
```

```
return 0;
}
```

Pour ma part, le choix est vite fait, je choisis `boost::assign`

 Cette méthode de concaténation (et plus généralement `boost::assign`) n'est pas applicable qu'aux conteneurs de la STL mais à tout ceux qui sont compatibles avec les notions qu'elle a introduites. C'est clairement ce qui se passe ici. `boost::array` offre une interface STL-like, elle est donc utilisable avec `boost::assign`.

Remplir un tableau d'objet constant de manière lisible.

```
#include <vector>
#include <iostream>
#include <string>
#include <boost/assign.hpp>

using namespace boost::assign;
using namespace std;

class S
{
    int a;
    std::string b;

    friend std::ostream& operator<<(std::ostream& flx, const S& s);
public:
    S(int a_, std::string b_):a(a_),b(b_){}
};

std::ostream& operator<<(std::ostream& flx, const S& s)
{
    flx<<s.a<<" "<<s.b;

    return flx;
}

int main()
{
    std::vector<S> array1;

    push_back(array1)(1,"toto")(2,"titi")(3,"tata");

    int size=array1.size();

    cout<<"s: "<<size<<endl;
    for(int i=0;i<size;i++)
    cout<<array1[i]<<endl;

    return 0;
}
```

Initialiser un vecteur d'objet constant: Une première idée être peut être tout simplement :

```
//ici S est la classe de l'exemple précédant.
const std::vector<S> array1= list_of(1,"toto")(2,"titi")(3,"tata");
```

Mais ce code ne va pas marcher. En effet, votre vecteur attend (et donc list\_of) attend des objets de type S. Or ici, on lui passe un int et une std::string. Il faut passer par des objets complets à list\_of. Ainsi le code devient:

```
const std::vector<S> array1= list_of(S(1,"toto"))(S(2,"titi"))(S(3,"tata"));
```

## IV - Conclusion

En conclusion, *boost::assign* est une petite partie de *boost* qui a pour but de vous simplifier la vie en vous rendant la tâche d'affectation plus facile. Mais il faut savoir elle risque de devenir obsolète avec les avancées de C++0x dans ce domaine, comme le prouve [cette vidéo](#), conférence de B. Stroustrup sur les *initializer lists* de C++0X. Enfin cette facilité actuelle est considérée par certaines personnes, à tort ou à raison, comme de l'obfuscation.

**Remerciements:** Je tiens à remercier Luc Hermitte ainsi que Alp pour leurs conseils et soutiens.

