

Template rebinding en C++

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

Date de publication : 26/07/2007

Dernière mise à jour :

Cet article va vous présenter le *template rebinding*, technique utilisée en C++ grâce aux *templates*, qui peut paraître assez obscure au premier abord.

- I - Qu'est-ce que le template rebinding ?
- II - Un exemple d'utilisation du template rebinding
- III - Conclusion
- IV - Remerciements

I - Qu'est-ce que le template rebinding ?

Derrière ce nom qui peut sembler compliqué, se cache une technique pourtant très simple. Analysons son nom.

Décomposition du nom template rebinding

- **template** : renvoie aux modèles (*templates*), permettant la programmation générique en C++
- **rebinding** : idée de réacheminer - on dispose de quelque chose puis à travers un certain lien, on renvoie cette chose, en la modifiant dans notre cas

Je vais avec regret couper court à vos interrogations et vous expliquer enfin en quoi consiste cette technique.

Imaginons que nous disposons d'une classe *template* S, comme suit.

Modèle de classe S

```
template <typename T>
struct S
{
    /* code */
};
```

Essayez de répondre à la question suivante tout seul.



Comment, à l'intérieur de S, pouvons-nous disposer d'un type S<U>, où U est un type quelconque ?

Peut-être vous est venue l'idée d'un modèle de définition de type (*template typedef*) ? La solution serait alors la suivante.

Template typedef à l'intérieur de S

```
template <typename T>
struct S
{
    template <typename U> typedef S<U> OtherType;
};

// On y accéderait ainsi
S<int>::OtherType<double> sd; // sd est de type S<double>
```

Ainsi, on aurait "réacheminé" le paramètre *template* de S en un autre type. La notion en question est bien la notion de *rebinding* exposée plus haut, appliquée à notre problème.

Il n'y a qu'un seul problème avec le code écrit ci-dessus : les *template typedefs* n'existent pas en C++. Cependant, l'introduction de ces derniers dans la prochaine version du C++ est bien entamée. Vous trouverez dans la conclusion des liens concernant le sujet.

Nous n'avons donc pas résolu notre problème. Il y a cependant un article dans la [FAQ FAQ C++](#) qui va nous aider à résoudre le problème : [FAQ Peut-on faire des alias sur des templates ?](#) Je vous invite au passage à consulter le lien donné en fin d'article de FAQ sur le sujet. Appliquons donc la technique donnée à notre cas. Nous voulons remplacer l'hypothétique code suivant.

Alias sur un template

```
template <typename U> typedef S<U> OtherType;
```

En appliquant l'astuce citée plus haut, on obtient le code suivant.

Astuce simulant un template typedef

```
template <typename T>
struct S
{
    /* du code */
    template <typename U>
    struct rebind
    {
        typedef S<U> OtherType;
    };
};
```

On constate maintenant que l'on peut utiliser le *template rebinding* comme suit.

Utilisation du template rebinding

```
#include <typeinfo>

template <typename T>
struct S
{
    typedef S<T> Type;

    template <typename U>
    struct rebind
    {
        typedef S<U> OtherType;
    };
};

int main()
{
    typedef S<int> Si;
    typedef Si::rebind<double>::OtherType Sd;
    Si a; Sd b;
    assert(typeid(a) != typeid(b)); // aucun problème
    return 0;
}
```

Seule ombre au tableau : on est obligés de passer par la structure interne *rebind* et son *typedef* interne "OtherType" pour arriver à nos fins. Cependant, cela n'est pas si contraignant que cela.

Nous avons donc présenté une technique simple mais efficace résolvant le problème que nous avons posé en début d'article. Cependant, tout cela demeurant assez abstrait, nous allons aborder en deuxième partie un exemple concret où l'utilisation de cette technique s'avère décisive.

II - Un exemple d'utilisation du template rebinding

Il s'agit maintenant d'imager notre *template rebinding* par un exemple concret.

Considérons un allocateur. C'est une classe/structure *template* de la forme suivante.

Allocateur

```
template <typename T>
struct allocator
{
    // fonctions qui allouent et désallouent la mémoire
};
```

Si l'on veut écrire un allocateur pour un tableau contenant des variables de type T, il suffit d'allouer la taille d'un T pour chaque "case" du tableau. C'est ce qu'utilise `std::vector` par exemple. Cependant si l'on veut écrire un allocateur pour une liste, comme `std::list`, il nous faut allouer deux types d'éléments :

- 1 les éléments contenus dans la liste (si c'est une `list<T>`, il faut allouer des T);
- 2 les *nodes* de la liste (une liste est en effet constituée de noeuds, où *nodes* en anglais).

Si l'on veut écrire un allocateur qui fonctionne pour une liste, il faut par conséquent à l'intérieur de notre classe ou structure `allocator` pouvoir allouer les deux types d'éléments, comme écrits plus haut. C'est pourquoi il nous faut disposer d'un second type, qui allouera les *nodes*. On obtient alors un code ressemblant à celui qui suit.

La struct Allocator revisitée

```
template <typename T>
struct allocator
{
    /* fonctions allouant/désallouant la mémoire */

    template <typename U>
    struct rebind
    {
        typedef allocator<U> other;
    };
};
```

On peut ainsi gérer en interne (dans `allocator`) l'allocation du type contenu dans la liste et les noeuds de la liste, puisque de plus l'allocateur connaîtra le type contenu dans la liste - qui remplacera T ci-dessus - ainsi que le type des noeuds de la liste, dépendant de T - qui remplacera U ci-dessus.

Nous avons donc vu un champs d'application précis de cette technique qui, je l'espère, vous a fait comprendre l'utilité du *template rebinding* dans un code C++.

III - Conclusion

Voici une liste de liens qui complètent ou aident à la compréhension de cet article.

-  http://c.developpez.com/faq/cpp/?page=templates#TEMPLATES_typedef
-  http://groups.google.fr/group/fr.comp.lang.c++/browse_thread/thread/dcb4df2d6102409b
-  <http://www.ddj.com/dept/cpp/184403759>
-  <http://angelikalanger.com/Conferences/Slides/CppPolicyMixins-CppWorld-1998.pdf>

IV - Remerciements

Merci à **LLB** et **hiko-seijuro** pour la relecture et correction de cet article.

