

# Présentation des classes de Traits et de Politiques en C++

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

Date de publication : 16/07/2007

Dernière mise à jour :

Cet article a pour but de vous présenter ces notions afin que vous puissiez les utiliser correctement dans vos applications, ce qui rendra vos structures bien plus flexibles et puissantes.

## I - Traits

- 1.1 - Qu'est-ce qu'une classe de Trait?
- 1.2 - Exemples de traits - Les plus utilisés
- 1.3 - Un peu de théorie

## II - Politiques

- 2.1 - En quoi consiste une classe de Politiques
- 2.2 - Utilisation avancée

## III - Conclusion

## IV - Remerciements

## I - Traits

### 1.1 - Qu'est-ce qu'une classe de Trait?

Une classe de trait est une classe (ou structure) qui associe à un type donné d'autres types (grâce à des *typedef*) ainsi que des fonctions membres *statiques*. La puissance des traits est due au fait que cela ajoute un niveau d'abstraction et permet d'ajouter un niveau de généricité. Pour imaginer un peu cette notion, regardons le code suivant.

#### Classe de Trait TypeDescriptor

```
template <typename T>
struct TypeDescriptor
{
    typedef T type;
    typedef T* pointer;
    typedef T& reference;
    typedef const T const_type;
    // ...
};

// Plus loin dans le code
int i = 42;
TypeDescriptor<int>::pointer pi = &i;
*pi = 24;
```

Quel que soit le type que l'on passera, les typedef résultants seront transparents.

Nous allons désormais voir une autre classe de trait, et utiliser la spécialisation. Il s'agit d'écrire une classe de trait permettant de savoir si le type passé est le type "int". Commençons par écrire la structure *template is\_int<T>*.

#### is\_int<T>

```
template <typename T>
struct is_int
{
    static const bool value = false;
};
```

Il n'y a à l'évidence qu'un seul moyen de faire que value soit vraie pour le type *int* : **la spécialisation**. Voici donc la spécialisation de notre structure *template*.

#### Spécialisation de is\_int

```
template <>
struct is_int<int>
{
    static const bool value = true;
};
```

Il est important de comprendre que la spécialisation est un outil très important lors de la création de classes de traits. Revenons sur notre exemple de TypeDescriptor pour mieux le comprendre.

Le code de TypeDescriptor écrit plus haut a l'air juste. On voit cependant un problème apparaître. Considérons le code suivant.

### Problème avec l'utilisation de TypeDescriptor

```
class A;
A myinstance;
TypeDescriptor<A&&>::reference ra = myinstance;
/* Problème!
 * Ce code équivaut à A&& ra = myinstance ,
 * ce qui est syntaxiquement invalide.
 */
```

Un mot clé : la **spécialisation**. Il suffit de spécialiser TypeDescriptor pour les références, comme suit.

### Spécialisation de TypeDescriptor

```
template <typename T>
struct TypeDescriptor<T&&>
{
    typedef T& type;
    // ..
    typedef T& reference;
    // ..
};
```

Ainsi, le code posant problème ne le fait plus. On pourra maintenant utiliser de manière transparente cette classe traits sans avoir une quelconque erreur. Ce qu'il est important de ressentir est que dans une structure ou une fonction *template*, on pourra utiliser cette classe traits en donnant à TypeDescriptor le type T dont on se sert dans la fonction (ou structure) *template* dont il est question, et le code ne provoquera alors pas d'erreur d'exécution et de compilation, sans même connaître le type T.

On voit donc que l'on vient d'introduire une couche dont je vous parlais plus haut, à la fois une couche d'abstraction et de généricité, qui permet une flexibilité accrue dans l'élaboration de nos codes. De plus, il est important de savoir que la spécialisation est particulièrement utilisée avec les traits car on peut d'une façon transparente optimiser notre code selon le type passé sans que le "code client" ne s'en doute, et comme ci-dessus, rendre notre code utilisable pour tous les types, et ce de manière transparente.

Les traits sont déjà beaucoup utilisés, et nous allons voir qu'il existe des traits que l'on utilise sans que l'on en ait forcément conscience.

## 1.2 - Exemples de traits - Les plus utilisés

La STL utilise la notion de traits. En effet, la liste suivante permet d'entrevoir l'utilisation de traits dans la STL, en donnant deux noms de Traits que l'on utilise sans même s'en douter lorsque l'on manipule la STL (respectivement avec les chaînes de caractères et itérateurs sur des séquences).

### Traits utilisés dans la STL

- Traits de caractères :  [http://www.sgi.com/tech/stl/char\\_traits.htm](http://www.sgi.com/tech/stl/char_traits.htm)
- Traits pour les itérateurs  [http://www.sgi.com/tech/stl/iterator\\_traits.html](http://www.sgi.com/tech/stl/iterator_traits.html)

Les deux liens fournis nous font comprendre, avec le paragraphe précédent, que les Traits sont une notion fondamentale de la programmation générique. De plus, vous êtes-vous douté durant vos premiers pas avec la STL qu'il existait de telles classes dans les rouages de la STL, qui vous paraissaient si complexes à l'époque ?

Il existe d'autres traits très utilisés. Il existe un regroupement de traits dans une bibliothèque faisant partie de Boost ([FAQ Qu'est-ce que Boost](#) et [Introduction à Boost par Miles](#)) permettant énormément de choses : [Boost.TypeTraits](#).


Vous pouvez consulter [à cet endroit](#) la liste exhaustive des classes de traits offertes par boost. Pour un type donné, on peut grâce à cette bibliothèque obtenir une référence vers ce type, obtenir un pointeur vers ce type, savoir si ce type a un destructeur virtuel, savoir si ce type est une classe, ... On peut donc manipuler très facilement des types et écrire des codes optimisés tout aussi facilement à l'aide de ces outils. Pour apprendre à s'en servir, je vous invite à consulter [la documentation de Boost.TypeTraits](#). Une chose importante à préciser : beaucoup des bibliothèques de Boost définissent et utilisent des traits, comme [Boost.Serialization](#) ou [Boost.Graph](#), ce qui peut vous servir d'exemple "concret" d'utilisation des traits.

Ceci clôt la liste des traits les plus utilisés que j'ai souhaité mentionner dans cet article. Vous pouvez cependant en trouver sur internet par le biais de recherches, si vous n'êtes pas encore satisfait.


### 1.3 - Un peu de théorie

À ce stade de l'article, vous devez avoir compris la notion de traits. Ce paragraphe va aborder un aspect plus *théorique* de la notion de traits.

Rappelons ce qu'est une classe de traits.

 *Une classe de traits, généralement template, définit des caractéristiques ou des fonctions associées à un type donné. Cela permet donc d'ajouter de l'information à des types que l'on ne peut pas modifier.*

Lorsque l'on analyse ce que l'on ne peut pas mettre à l'intérieur d'une classe de traits, on obtient donc tout ce qui est non-statique.

 *Une classe de trait ne doit pas posséder de membre non-statique.*

Posons-nous désormais la question suivante : comment utilisera-t-on une classe de traits ? Voyons le code suivant.

#### Un exemple de trait : `is_pointer`

```
template <typename T>
struct is_pointer
{
    static const bool value = false;
};

template <typename T>
struct is_pointer<T*>
{
    static const bool value = true;
};
```

La structure `template is_pointer`, qui est une classe de traits, permet simplement de déterminer si le type passé en paramètre est un pointeur sur un autre type. Ce qui permet de le savoir dans le code, c'est l'attribut statique constant "value", de type `bool`, qui est accessible à la **compilation**.

Et que doit-on écrire quand on veut s'en servir? Observez le code suivant.

#### Utilisation de `is_pointer<T>`

```

template <bool is_a_pointer>
void f()
{
    std::cout << "Je ne sais pas qui je suis" << std::endl;
}

template <>
void f<true>()
{
    std::cout << "Je suis un pointeur!" << std::endl;
}

struct Foo
{
    template <typename T>
    void Bar()
    {
        f< is_pointer<T>::value >();
        /* affichera "Je ne sais pas qui je suis" si T n'est pas un pointeur et
        "Je suis un pointeur!" dans le cas contraire */
    }
};

// ...
Foo f;
f.Bar<int>(); // affiche "Je ne sais pas qui je suis"
f.Bar<int*>(); // affiche "Je suis un pointeur!"
    
```

Nous voyons que le fait de savoir si `T` est un pointeur va nous permettre de modifier le code appelé grâce à la spécialisation de `f`, ce qui peut *par exemple* permettre d'optimiser le code, en donnant une version optimisée uniquement pour les pointeurs, et une version générique pour les autres types, moins optimisée.

Comme on peut le voir dans `Boost.TypeTraits`, il est possible ainsi d'obtenir énormément d'informations sur les types passés en paramètre de nos codes génériques, afin d'optimiser grâce à la **spécialisation** notre code selon le type passé.

 *Quel est le rapport avec cette transparence d'utilisation ?*

Le rapport est simple. Lorsque vous utiliserez `is_pointer<T>` dans un code générique, `T` pourra être à priori n'importe quel type, car votre code sera une structure ou une fonction *template*. Vous pourrez donc spécialiser votre code selon le cas où `T` est un type de pointeur ou non, en général dans le but d'optimiser les performances ou pour éviter des erreurs de syntaxe cachées dans votre code. Cependant, lorsque l'utilisateur se servira de votre code, en passant les types dont il se sert en paramètre à votre code *template*, il n'aura rien à faire en plus : son code ne provoquera aucune erreur de compilation et selon votre code, des optimisations seront apportées par rapport à une version unique générique de votre code.

Certains d'entre vous risquent cependant de trouver tout cela encore trop abstrait. J'entends par là qu'ils se posent peut-être la question suivante : "Dans notre code de tous les jours, ça nous servira à quoi les traits ?". Je vais présenter un exemple plus concret, suggéré par **Laurent**, qui vous fera plus facilement comprendre l'utilité et la puissance des classes de traits. Merci à lui.

Rentrons dans le contexte. On veut écrire une fonction `min` pour comparer deux objets du même type et retourner le plus petit. A l'évidence, pour comparer des entiers, il vaut mieux les prendre par valeur, alors que pour des `std::string`

par exemple, une référence constante permettra une comparaison plus rapide. Ecrivons donc une classe de traits qui nous retourne le meilleur type possible à passer en argument de notre fonction min. Si la taille de T est plus grande que 8, on prend une référence constante. Ceci n'est pas parfait, mais c'est simplement pour illustrer l'utilisation des classes de traits dans un code que l'on peut écrire au quotidien. Cela donne le code suivant.

#### Écriture d'une classe de traits CallTraits

```
template <typename T>
struct CallTraits
{
    template <typename U, bool Big> struct CallTraitsImpl;

    template <typename U>
    struct CallTraitsImpl<U, true>
    {
        typedef const U& Type;
    };

    template <typename U>
    struct CallTraitsImpl<U, false>
    {
        typedef U Type;
    };

    typedef typename CallTraitsImpl<T, (sizeof(T) > 8)>::Type ParamType;
};
```

Ainsi, le type ParamType est le "meilleur" type à prendre en argument de min pour optimiser notre fonction. Voici maintenant la définition de la fonction min.

#### Définition de min

```
template <typename T>
T Min(typename CallTraits<T>::ParamType X, typename CallTraits<T>::ParamType Y)
{
    return X < Y ? X : Y;
}
```

Et lors de l'appel de min pour différents types, l'appel est optimisé selon le type des objets que l'on compare.

#### Utilisation de min


```
std::string s1 = "Salut";
std::string s2 = "Bonjour";
std::string s3 = Min<std::string>(s1,s2); // passage par référence constante
// s3 vaut "Bonjour"

int i1 = 1;
int i2 = 2;
int i3 = Min<int>(i1,i2); // passage par valeur
// i3 vaut 1
```

Maintenant que vous avez intégré la notion de classe de traits, vous disposez d'une base vous permettant d'écrire un code bien plus souple, optimisé, sans avoir à en écrire énormément. Il ne faut cependant pas s'arrêter là. La lecture de code utilisant des classes de traits est importante afin d'approfondir sa compréhension et d'élargir le spectre d'utilisation de cet outil.

Il y a cependant des inconvénients lorsque l'on se sert de traits. En effet, on se rend compte que l'écriture de traits devient lourde et peut même paraître superflue. Il faut réfléchir à quand on doit en utiliser, et quand on ne doit pas. De plus, il faut faire attention lorsque l'on définit des spécialisations dans certains fichiers, en faisant attention à ne pas


inclure un fichier contenant une spécialisation avant le moment où l'on doit s'en servir, ce qui risquerait de modifier le comportement de votre code, et provoquer un comportement inattendu et imprévisible à première vue.


Pour en finir avec les traits, je souhaite préciser que les traits vont tendre à disparaître avec l'apparition des concepts. En effet ces derniers permettent d'obtenir des résultats identiques en écrivant un code plus simple, plus court et certainement plus clair. Pour ceux qui ignorent ce que sont les concepts, je les invite à lire  [ceci](#) ainsi que [FAQ cela](#) pour en découvrir plus sur la prochaine version du C++ qui utilisera les concepts.

## II - Politiques

### 2.1 - En quoi consiste une classe de Politiques

Pour citer [FAQ la FAQ C++](#), on peut définir les classes de politiques ainsi.

 *Les classes de politique (policy classes) sont assez similaires aux classes de traits, mais contrairement à celles-ci qui ajoutent des informations à des types, les classes de politiques servent à définir des comportements.*

Sur cette partie de la FAQ, on peut voir une citation de Andrei Alexandrescu qui est celui qui a réellement fait connaître la notion de classe de politique à travers le livre qu'il a écrit, et montre la puissance de cette notion dans la bibliothèque qu'il a écrite :  **Loki**.

Cependant, vous vous demandez probablement à quoi cela correspond de définir des comportements. Nous avons vu que les traits, eux, définissaient principalement des propriétés associées à des types. Autrement dit, les traits associent des informations à des types, principalement avec des attributs statiques et des *typedefs*, et de façon plus rare avec des fonctions statiques.

Par comportement, on entend bien évidemment comportement à l'exécution. La nature des classes de politiques ne sera pas tellement différente de la nature des classes de traits, mis à part le fait que les classes de politiques soient centrées sur le comportement, et que par conséquent il s'agira plutôt d'écrire des fonctions (souvent statiques) à l'intérieur de ces dernières. Penchons-nous sur le code écrit dans l'article de FAQ donné plus haut.


#### Introduction aux classes de politiques

```
template <typename T>
struct Addition
{
    static void Accumuler(T& Resultat, const T& Valeur)
    {
        Resultat += Valeur;
    }
};

template <typename T, typename Operation>
T Accumulation(const T* Debut, const T* Fin)
{
    T Resultat = 0;
    for ( ; Debut != Fin; ++Debut)
        Operation::Accumuler(Resultat, *Debut);

    return Resultat;
}
```


Afin de mieux pouvoir parler de ce code, il s'agit de mettre des mots sur certaines notions sous-jacentes dans ce code.

 *Une Politique définit une interface de classe ou une interface de classe template.*

Lorsque l'on se donne une politique, on peut l'implémenter d'une infinité de manières différentes.

 *Les implémentations d'une politique sont appelées classes de politique.*

Dans le code donné plus haut, Operation est une politique, dont l'interface est composée d'une fonction statique Accumuler. La structure *template* Addition définit une implémentation de la politique Operation : c'est une classe de politique. Toute classe définissant une fonction statique Accumuler dont la signature convient est une classe de politique conforme à la politique Operation.

 Les classes qui utilisent au moins une politique sont appelées classes hôtes.

Soit une classe hôte H donnée. Soient  $P_1, \dots, P_n$  les politiques définies par H. Pour instancier H, il faut pour chaque  $P_k$  fournir une classe de politique conforme à  $P_k$ , c'est à dire qui propose une implémentation complète de la politique  $P_k$

Ceci nous montre qu'une classe hôte est en fait conçue afin d'assembler toutes les politiques (en réalité, leurs implémentations), pour les unir dans une unité complexe. Plus ces politiques sont indépendantes, plus la flexibilité est accrue et les possibilités sont larges. Pour parler d'indépendance de politiques (ainsi que de leurs implémentations), on parle généralement d'*orthogonalité* de politiques. Si les politiques d'une classe hôte sont deux à deux orthogonales, alors la classe hôte en question est flexible et offre une diversité d'utilisation incroyable. En effet, dans ce cas, on peut "personnaliser" le comportement de cette classe à l'infini, le tout avec une petite quantité de code.

## 2.2 - Utilisation avancée

Il s'agit maintenant de découvrir des techniques améliorant l'utilisation de politiques. Nous allons en étudier deux. La première est l'ajout de fonctionnalités optionnelles, c'est à dire ajouter une fonction à la classe hôte, par exemple, que pour une certaine classe de politique. La deuxième est la modification de structure d'une classe.

Pour l'ajout de fonctionnalités optionnelles, un code sera plus explicite qu'un long paragraphe.

### Ajout d'une fonctionnalité optionnelle

```
// classe hôte
template <typename MyPolicy>
struct HostClass
{
    void f()
    {
        MyPolicy::Foo();
    }

    void g()
    {
        MyPolicy::Bar();
    }
};

struct P1
{
    static void Foo()
    {
        // fait quelque chose
    }
};

struct P2
{
    static void Foo()
    {
        // fait quelque chose
    }

    static void Bar()
};
```

#### Ajout d'une fonctionnalité optionnelle

```
{  
    // fait autre chose  
};
```

On remarque ici que `HostClass<MyPolicy>::g` ne sera appelée que si la classe de politique est `P2`, car sinon une erreur de compilation surviendra. Autrement dit, on dispose d'une fonction supplémentaire si la politique utilisée est `P2`. On a donc bien ajouté une fonctionnalité à `HostClass` pour une classe de politique donnée.

À présent, il s'agit de modifier la structure d'une classe selon la classe de politique utilisée. La solution de notre problème est simple : *l'héritage*. Cependant, on peut également utiliser la composition, mais on ne change pas réellement la structure, on en donne l'illusion. Encore une fois, un code sera plus parlant.

#### Modifier la structure d'une classe hôte


```
template <typename MyPolicy>  
struct HostClass : public MyPolicy  
{  
    // code de HostClass  
};  
  
struct P1  
{  
    void Foo1();  
    void Bar1();  
    typedef int MyIntegerType;  
};  
  
struct P2  
{  
    void Foo2();  
    void Bar2();  
    void FooBar();  
    typedef char MyCharType;  
};
```

Analysons ce qu'entraîne ce code. La structure `HostClass<P1>` possède le code de `HostClass`, avec en plus les fonctions `Foo1`, `Bar1` et un type `MyIntegerType`. Cependant, la structure `HostClass<P2>` possède également le code de `HostClass`, mais en plus elle dispose des fonctions `Foo2`, `Bar2`, `FooBar` et du type `MyCharType`, mais ne dispose pas des fonctions `Foo1` et `Bar1`, ni du type `MyIntegerType`. Selon la classe de politique utilisée, la structure de `HostClass` change quasi-totalement, au code présent dans `HostClass` près. Encore une fois, on peut obtenir un effet similaire à l'aide de la composition, mais c'est bien moins "naturel". Pourquoi? Hé bien il faut garder à l'esprit que l'héritage représente la relation *EST-UN*, et la composition représente la relation *EST-IMPLEMENTE-EN-FONCTION-DE*. La relation *EST-UN* est beaucoup plus forte, ce qui entraîne que l'on modifie naturellement la structure de `HostClass`, sans avoir à écrire de code supplémentaire, tandis qu'avec la composition, pour modifier la structure de `HostClass`, il aurait fallu écrire du code supplémentaire afin d'intégrer la structure de la classe de politique à la classe hôte. Cependant, faites bien attention à ne pas abuser de l'héritage.

Une dernière chose à propos de l'héritage doit être signalée ici. Lorsque l'on écrit une classe de politique, si la classe hôte hérite de cette dernière, on va bien évidemment modifier sa structure, mais on peut entre autres modifier son **interface**, c'est à dire ce qui est accessible en dehors de la classe - ce qui est vu par le reste du code. C'est une faculté qui peut s'avérer très utile.

Les politiques vous rappellent peut-être le design pattern *Strategy*, car en réalité c'est ce à quoi elles correspondent. Cependant, elles exploitent les possibilités du C++ que n'offrent certains autres langages, ce qui les rend cependant différentes d'implémentations du design pattern *Strategy* que l'on pourrait trouver en Java, par exemple.

Lorsque l'on écrit une classe hôte, il est important de faire en sorte d'utiliser le moins de politiques possibles tout en disposant de politiques orthogonales. Il ne s'agit pas de se priver de politiques, ou de faire un choix entre les politiques, mais plutôt de ne pas avoir de politiques superflues, dépendantes d'autres politiques.

 *Lorsque vous écrivez des classes hôtes, utilisez des politiques orthogonales et assurez-vous que chaque politique n'est pas superflue.*

Pour terminer, nous allons parler des inconvénients des politiques. À l'évidence, lorsque l'on utilise des politiques en C++, une certaine complexité de code s'installe. Ceci peut déstabiliser certaines personnes et les empêcher d'exploiter tout ce qu'offrent les politiques. Il y a un autre inconvénient "majeur" : l'explosion combinatoire. Il s'agit là d'une sorte de *déravage* lorsque l'on instancie une classe hôte en lui donnant en paramètre des implémentations de politiques bien définies. Cela peut parfois mener à des situations non envisagées. C'est pourquoi il faut être prudent lors de l'utilisation de politiques en C++, car il vaut mieux que l'utilisateur utilise une classe testée et maîtrisée plutôt qu'une classe qui peut mener à des comportements étranges et surtout non attendus, donc non prévus par votre code.

### III - Conclusion

C'est fini pour ce tutoriel, en espérant que vous avez pu découvrir de nouvelles choses, et que vous maîtrisez les notions abordées dans ce tutoriel. Voici quelques liens essentiels sur les sujets traités dans cet article.

- [FAQ Classe de Traits dans la FAQ C++](#)
- [FAQ Classe de Politique dans la FAQ C++](#)
-  <http://www.cantrip.org/traits.html>
-  [http://www.boost.org/more/generic\\_programming.html#policy](http://www.boost.org/more/generic_programming.html#policy)
-  [http://en.wikipedia.org/wiki/Policy-based\\_design](http://en.wikipedia.org/wiki/Policy-based_design)

## IV - Remerciements

Merci particulièrement à **Laurent Gomila**, **Loic Joly**, **rod**, **Miles**, **gege2061**, **LLB** et Gawen pour m'avoir relu et/ou m'avoir aidé à améliorer ce tutoriel.

