Dtails techniques

March 24, 2016

Contents

1 classe spcialisation

- Si il existe une spcialisation (complte ou partielle) qui match avec l'utilisation de la classe, cette spcialisation sera utilise.
- Si l'utilisation de la classe moins de type que dans la dclaration de la classe, les types par dfaut seront affect.

```
template <class T, class U = int >
struct demo
{
    constexpr static int value = 0;
};

template<class T>
struct demo <T,int>
{
    constexpr static int value = 1;
};

template<class T>
struct demo <T,void>
{
    constexpr static int value = 2;
```

```
};

int main() {
    using namespace std;
    cout << demo<std::string>::value << endl;
    cout << demo<int,int>::value << endl;
    cout << demo<int,void>::value << endl;
    cout << demo<int,std::string>::value << endl;

}
```

Va afficher : "1 1 2 0". Les rponses 2 et 3 sont sans surprise l'utilisation de la spcialisation. La rponse 4 est le cas ou aucune spcification ne match et la rponse 1 montre que lorsqu'on qu'un seul argument template, les valeurs par dfauts sont appliques puis la slection de la classe la plus spcialis se fait. Ainsi dans cet exemple tout appel "demo" avec un paramtre matchera avec la spcification ¡T,int¿

2 SFINAE

Le principe de SFINAE : "Substitution failure is not an error". Lorsque le compilateur rsous un type ou un appel de fonction il veut au final avoir exactement un type/fonction a appeler (une fois les regles de priorit appliqu s'il y en a). S'il y en plus il ne saura pas quoi choisir, et s'il n'y en a pas il ne pourra rien faire.
Ce que SFINAE dit c'est "Si tu trouves quelque chose qui match, regarde si la signature + type de retour un sens en terme de type, si c'est pas le cas, c'est pas grave, passe au suivant".
Avec un exemple sur une fonction (ce n'est pas jsute de la surcharge de fonction car toutes ces fonctions peuvent tre appel avec un T)

On notera dans le code la prsence de "typename" lorsqu'un utilise un type d'une classe template "T::monType". Comme la syntaxe "T::qqch" peut aussi bien etre un type qu'un attribut statique, le compilateur nous demande de prciser avec "typename" lorsqu'il s'agit d'un type.

```
template <class T>
typename T::value_type fonction_sfinae (const T& t){
    std::cout << "Le type T::value_type a un sens " << std::endl;
    return typename T::value_type(); // je supose qu'il est default
    // constructible mais c'est juste pour l'exemple.
}

template <class T>
void fonction_sfinae(T t, typename T::bibi d = 1) {
```

```
     std::cout << "Le type T::bibi a un sens " << std::endl;
}

template <class T, class U = typename T::boo >
void fonction_sfinae(T t) {
     std::cout << "Le type T::boo a un sens " << std::endl;
}

struct booclass {
    using boo = int;
};

struct bibiclass {
    using bibi = int;
} ;

struct drame {
    using bibi = int;
    using boo = int;
} ;
int main() {
    using namespace std;

    fonction_sfinae(vector<int>());
    fonction_sfinae(booclass ());
    fonction_sfinae(bibiclass ());
    /*fonction_sfinae(drame ());
    |39|error: call of overloaded  fonction_sfinae(drame)
    is ambiguous|
    |39|note: candidates are:|
    |12|note: void fonction_sfinae(T, typename T::bibi)
    [with T = drame; typename T::bibi = int]|
    |17|note: void fonction_sfinae(T) [with T = drame; U = int]|*/
    // fonction_sfinae(5); |44|error: no matching function for
     call to  fonction_sfinae(int)   |
}
```

L'affichage sans surprise :
Le type T::value_type a un sens
Le type T::boo a un sens
Le type T::bibi a un sens
Pour le cas de "drame" on a comme le compilateur nous l'indique, deux fonctions qui match. Il ne peut pas choisir. Et le cas de l'entier est le cas ou personne ne match.

3 decltype, declval, enbale_if et autre joyeuset

decltype : permet de retourner le type d'une expression, a saute pas forcement au yeux comme a, mais c'est gnial.

```
        int a = 4 ;
        decltype(a) b = 6; // a est un int
        decltype(monObjet.maMethode()) k;

        template<
            class T,
            class U = decltype(T().begin()),
            class W = decltype(T().end())
        >
    ....
```

Le fonctionnement est assez proche de auto, mais les rgles ne sont pas exactement les "const" et les rfrence. (google "auto vs decltype vs decltype(auto)").

declval : rpond au probleme suivant : Si on veut connatre le type de retour d'un mthode donc doit faire un decltype sur l'appel de la mthode. Mais comment fait on lorsque le type n'est pas default constructible ? class W = decltype(T().end()) // marche si T() est valide class W = decltype(std::decltype¡T¿().end()) // marche.
enbale_if : enable_if¡false¿::type n'existe pas et enable_if¡true¿::type existe (et a vaut void)
same_if : same_if¡T,U¿::value vaut vrais si T==U et faux si non.

A noter :

```
struct A {
        constexpr static bool value = true;
};
 quivaut
struct A : std::true_type{}
```

4 Application, affichage des conteneurs de la STL

Avec a la spcialisation, SFINEA et ce qu'on vient de voir on peut cre notre premier type trait.

```
template <class T, class U = void, class V = void>
struct is_container : std::false_type {};

template <class T>
```

```
struct is_container<   T,
                                    decltype(std::declval<T>().begin()),
                                    decltype(std::declval<T>().end()
            >   : std::true_type {};
```

Et la ... a ne marche pas :D. Pourquoi ?

- Si T n'est pas un conteneur, la spcialisation n'est pas pris en compte car elle n'est pas valide.
- Si T est un conteneur on va cherche un match avec T void void. La spcialisation choue car elle spcialise T T::iterator T::iterator ou quelque chose dans se gout la.

On veut du void ! Dans se cas :

```
template<class T>
struct void_if_valide {
    using type = void;
};

template <class T, class U = void, class V = void>
struct is_container : std::false_type {};

template <class T>
struct is_container<   T,
                typename void_if_valide<decltype(std::declval<T>().begin(
                typename void_if_valide<decltype(std::declval<T>().end())
            >   : std::true_type {};
```

Avec a on peut (relativement) simplement afficher tous les conteneurs de la STL (il faudra faire operator ¡¡ pour paire si on veut aussi afficher les map). On prendra soins que notre oprateur ne match pas les string et les wstring pour lesquels l'operateur ¡¡ existe dj.

```
template<class T>
struct void_if_valide {
    using type = void;
};

template <class T, class U = void, class V = void>
struct is_container : std::false_type {};

template <class T>
struct is_container<   T,
                typename void_if_valide<decltype(std::declval<T>().begin(
                typename void_if_valide<decltype(std::declval<T>().end())
```

```
            >  : std::true_type {};

template <class T, class U = void>
struct is_string : std::false_type {};

template <class T>
struct is_string<   T,
                typename void_if_valide<typename std::enable_if<
                                                                    std::is_sa
                                                                    std::is_sa
                                                                    >::type
                                                >::type
            >  : std::true_type {};



template<
        class T,
        class V = typename std::enable_if< is_container<T>::value >::type
        class U = typename std::enable_if< !is_string<T>::value >::type
      >
std::ostream& operator<< (std::ostream& out, const T& container)
{
    for (const auto& val : container)
    {
       // out << "\t" << val << "\n" ;
       out << val << " " ;
    }
    return out;
}
```