Déduction des types des paramètres

MartinMorterol

28 mars 2016

Sommaire

Je vais présenter deux cas d'utilisations :

- Le premier est celui annoncer dans le résumé du talk. Stoker une fonction passé en tant que template dans une std : :function.
- Le deuxième est faire un message d'erreur propre lors d'un mauvais passage de fonction.

Contexte : Un petit jeu en 2D.

- Une carte est un conteneur au sens de la STL.
- On peut donc l'utiliser dans un for range.

Problème :

- Une carte contient des std : :reference_wrappeur<Case>.
- J'ai pas envie que l'utilisateur connaisse ma structure interne ni qu'il en dépende.
- J'ai pas envie d'écrire .get() à chaque fois.
- J'ai envie de me faire plaisir ça sert à ça les projets perso :)

Solution : Je crée mon iterateur maison que j'appelle FoncteurIterator.

FoncteurIterator.hpp
Problème :

- Pour utiliser un std : :function<> J'ai besoin du type de retour du foncteur et du type du paramètre.
- Je ne voulais pas juste utiliser "FoncteurTemplate foncteur" (bien que ça soit le seule solution valide en C++14) pour me forcer à utiliser des choses que je ne connaissait pas.

Solution :

- FoncteurIterator< IteratorTemplates, FoncteurTemplate, typeRetour, typeParam>, Ok mais j'aime pas
- Trouver un moyen de déduire typeRetour et typeParam directement de FoncteurTemplate.

Pour la deuxième application je vais partir de "std : :find_if" et montrer comment on peut améliorer les messages d'erreur.
Les messages dans se cas ne sont pas horrible mais l'amélioration des messages d'erreur est toujours un plus, surtout en méta-programmation.

Sommaire

Je vais présenter les techniques que j'ai utilisé puis vous présenter la solution en elle même. Il n'y a pas de grosse difficulté mais ça peut faire beaucoup de chose nouvelles pour ceux qui n'ont jamais fait de méta-programmation.

Sommaire

La spécialisation (partielle ou non) est le fait de, pour une structure, spécifier une partie (ou tous) des types. Dans se cas le compilateur choisir la structure la plus spécialiser.

```
template< class T > struct is_pointer_helper
 : std::false_type {};
template< class T > struct is_pointer_helper<T*>
: std::true_type {};
```

source : cppreference

Si la structure à des types par défaut ils sont déduit avant de regarder quelle structure est la plus spécialisé (=> 1 1 2 0)

```
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; };

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;
```

Sommaire

Le principe de SFINAE (Substitution failure is not an error) est que si il y a un type qu'on n'arrive pas à résoudre on ignore simplement la déclaration. Pour les fonctions on peut utiliser le type de retour et les arguments en plus des paramètres template pour faire du SFINAE.
sfinae.cpp

Sommaire

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

```
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())
    >
....
```

declval : répond au probleme suivant : Si on veut connaître le type de retour d'un méthode donc doit faire un decltype sur l'appel de la méthode. 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. enbale_if prend un deuxième argument optionnel qui donne le type de " : :type", par défaut il vaut void.
same_if : same_if<T,U> : :value vaut vrais si T==U et faux si non.

Sommaire

On a globalement tout ce qui nous faut. On notera juste qu'hériter de std : :true_type ou std : :false_type nous permet de déclarer rapidement un "constexpr static bool value=true/false ;"
Exemple : traits.cpp

Sommaire

Ce code ne marche pas : error : redefinition of template<class T, class U> void foo(const T&)

```
template< class T,
  class U =  typename std::enable_if
    < is_container<T>::value  >::type >
void foo (const T& t) {
    std::cout << "je match  les conteneurs" << std::endl;
}
template< class T,
          class U =  typename std::enable_if
                     < !is_container<T>::value  >::type >
void foo (const T& t) {
    std::cout << "je match pas les conteneurs" << std::endl
}
```

Ce code marche, mais le passage à l'échelle est mauvais :/

```
template< class T,
  class U =  typename std::enable_if
   < is_container<T>::value  >::type >
void foo (const T& t) {
    std::cout << "je match  les conteneurs" << std::endl;
}
template< class T,
  class U =  typename std::enable_if
   < !is_container<T>::value  >::type,
          class W =  typename std::enable_i
           f< !is_container<T>::value  >::type>
void foo (const T& t) {
    std::cout << "je match pas les conteneurs" << std::endl
}
```

Celui la marche, passe à l'échelle, mais la lecture du type de retour est moins directe.

```
template< class T >
typename std::enable_if<
              is_container<T>::value, void  >::type
 foo (const T& t) {
    std::cout << "je match  les conteneurs" << std::endl;
}
template< class T >
typename std::enable_if<
  ! is_container<T>::value , void  >::type
foo (const T& t) {
    std::cout << "je match pas les conteneurs" << std::endl
}
```

On utilise le fait que les fonctions templates les plus spécialisées ont la priorité.

```
template <class T> struct informationParam

template <typename ClassType, typename ReturnType,
          typename... Args>
struct informationParam<ReturnType(ClassType::*)(Args...) >

template <typename ClassType, typename ReturnType,
          typename... Args>
struct informationParam<ReturnType(ClassType::*)(Args...)
                        const>

template < typename ReturnType, typename... Args>
struct informationParam<ReturnType(*)(Args...)>
```

Le gros de la magie est fait, il reste à faire le traitement.

```
template<class ReturnType, class ... Args>
struct informationParamParamFactorisation{
    constexpr static size_t arity = sizeof...(Args);
    using  result_type = ReturnType;
    template <size_t indice>
    struct arg_type_{
        static_assert ((indice < arity ), "msg d'erreur" );
        using type= typename std::tuple_element<indice,
                        std::tuple<Args...>>::type;

    };
    template <size_t i> using arg_type =
                                typename arg_type_<i>::type;
};
```

Maintenant on prend soin de nos utilisateurs en mettant des erreurs claires.

```
template <class... T>
class ERREUR;

// Le cas general si on arrive la c'est qu'on a perdu,
//  on compile pas et on informe l'utilisateur
//  que "ERREUR<LeParamNestPasUneFonction>"

template <class T>
struct informationParam{
    struct LeParamNestPasUneFonction {};
    ERREUR<LeParamNestPasUneFonction> erreur;
};
```

Si on résume, ça nous donne : paramInfo.hpp

Sommaire

La fonction qui donne le nombre d'argument, j'ai pas réussi à faire un beau message d'erreur, du coup j'ai mis un commentaire la ou le compilateur va envoyer l'utilisateur. Mieux que rien...

```
template<class T>
constexpr size_t nbParam(T fonction){
    // si ça plante ici c'est que ta classe elle a pas
                                    d'operateur () , noob
    return decltype(getInformationParam(fonction))
                                            ::type::arity;
}
template<class T> constexpr size_t nbParam(){
    return decltype(getInformationParam(std::declval<T>()))
     ::type::arity; // idem (pas de place dans le slide)
}
```

La fonction principale, avec deux écritures en fonction de ce qui est le plus simple pour l'utilisateur.

```
template<size_t nb,class T>
constexpr typename decltype(getInformationParam(
    std::declval<T>())):::type::template arg_type<nb>
    typeParam(T fonction );

template<size_t nb,class T>
constexpr typename decltype(getInformationParam(
     std::declval<T>())):::type::template arg_type<nb>
     typeParam();
```

Et puisque que ça me coûte rien je fais le même avec le type de retour.

```
template<class T>
constexpr typename decltype(getInformationParam(
    std::declval<T>()))::type::result_type
    typeRetour(T fonction );

template<class T>
constexpr typename decltype(getInformationParam(
    std::declval<T>()))::type::result_type
    typeRetour( );
```

Questions ?