Déduction des types des paramètres

Martin Morterol

30 mars 2016

Sommaire

Cas d'utilisations :

- Le premier est celui annoncé dans le résumé du talk. Stocker 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 :

- Solution Ok mais moche :

```
FoncteurIterator< IteratorTemplates,
    FoncteurTemplate,typeRetour, typeParam>
```

- Trouver un moyen de déduire typeRetour et typeParam directement de FoncteurTemplate.

Pour la deuxième application :
Problème :

- Les erreurs des méthodes attendant un "callable" sont vite moche.
- Exemple avec `std::find_if`.

Solution :

- Encapsuler `std::find_if`.
- Trouver un moyen de faire de `static_assert` sur le type de retour et les paramètres.

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

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 du code : cppreference

Si la structure à des types par défaut ils sont déduit avant de regarder quelle structure est la plus spécialisé

```
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;      // 1
cout << demo<int,int>::value << endl;          // 1
cout << demo<int,void>::value << endl;         // 2
cout << demo<int,std::string>::value << endl; // 0
```

Sommaire

- Substitution failure is not an error : Si on arrive pas à résoudre la déclaration, on l'ignore.
  - Pour les structures : Dans les type template et les types de spécialisation.
  - Pour les fonctions : Dans les types template, les arguments, et le type de retour.
- Exemple : 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 val
class W = decltype(std::declval<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.

`is_same` : `is_same<T,U>::value` vaut vrais si T==U et faux si non.

Sommaire

- Hériter de `std::true_type` ou `std::false_type` nous permet de déclarer rapidement un
  `constexpr static bool value=true/false;`
- On a tout ce qui nous faut !

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::en
}
```

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 V = void >
void foo (const T& t) {
    std::cout<<"je match pas les conteneurs"<<std::en
}
```

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::en
}
```

type dispatching

```
template < class T>
void foo (T&& t) {
    foo(std::forward<T>(t), is_container<T> ());
}

template < class T>
void foo  (T&& t , std::true_type  )
{
    std::cout<<"je match les conteneurs"<<std::endl;
}
template < class T>
void foo  (T&& t , std::false_type   )
{
    std::cout<<"je match pas les conteneurs"<<std::en
}
```

Fonction d'aide

On peut utiliser simplifier l'écriture du code utilisateur que ça soit au niveau du type de retour ou grâce à la déduction automatique du type.

```
decltype(typeRetour(lambda))
vs
decltype(getInformationParam(lambda))::type::
                                        result_type>::value
```

Sommaire

On stocke simplement les informations que l'on veut

```
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
};
```

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...)>
```

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

```
template <class... T>
class ERREUR;
template <class T>
struct informationParam{
    struct IsNotACallable {};
    ERREUR<T,IsNotACallable> erreur;
};
vs
template <class T>
struct informationParam{
    static_assert(
        std::is_same < T, typename void_if_valide<T>
                                ::type >::value &&fals
            "L'argument template n'est pas callable
            );};
```

Si on résume, ça nous donne : paramInfo.hpp

Sommaire

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 ?