Déduction des types des paramètres

Martin Morterol

1er avril 2016

Sommaire

Cas d'utilisations :

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

  Stocker une fonction passée 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 (comme `vector`, `map`...).
- On peut donc l'utiliser dans un *rang-based for loop*.

Problème :

- Une carte contient des `std::reference_wrappeur<Case>`.
- Je n'ai pas envie que l'utilisateur connaisse ma structure interne ni qu'il en dépende.
- Je n'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`.

Exemple : iterator.cpp
Exemple : iterateurFoncteur.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 la seule solution valide en C++14) pour me forcer à utiliser des choses que je ne connaissais 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 vites moches.
- 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 ce cas, ne sont pas horribles 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 ce cas le compilateur choisir la structure la plus spécialisé.

```
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 a des types par défaut ils sont déduits avant de regarder quelle structure est la plus spécialisée.

```
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; //
```

Si la structure a des types par défaut ils sont déduits avant de regarder quelle structure est la plus spécialisée.

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

`std::declval` : répond au problème 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 faire lorsque le type n'est pas *default constructible* ?

```
class W = decltype(T().end())// marche si T() est valide
class W = decltype(std::declval<T>().end()) // marche.
```

`std::enbale_if` : `enable_if<false>::type` n'existe pas et `std::enable_if<true>::type` existe. `enbale_if` prend un deuxième argument optionnel qui donne le type de "`::type`", par défaut il vaut void.

`std::is_same` : `is_same<T,U>::value` vaut vrai si T==U et faux sinon.

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 :
Pourquoi ?

```
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 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 V = void > // <-- +1 ligne par surcharge
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
//          Le vrais type est la --^^
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 peut aussi utiliser un paramètre par défaut, en utilisant un pointeur que l'on initialise à null.

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

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

Et enfin on peut utiliser du type dispatching :

```
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::endl;
}

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

Fonction d'aide

On peut simplifier l'écriture du code utilisateur 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
vs
type_result<decltype(lambda_test)>
```

Le dernier utilise les usings template, c'est sans doute le plus simple lorsqu'un utilise une classe "callable".

```
decltype(typeRetour<structParenthese>())
vs
type_result<structParenthese>
```

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,T>::value &&false,
            "L'argument␣template␣n'est␣pas␣callable"
            );
};
```

Si on résume, ça nous donne : paramInfo.hpp

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( );
```

Puisqu'on est chaud on peut regarde l'application au notre

`std::find_if`

source : main.cpp

Questions ?