Déduction des types des paramètres

MartinMorterol

8 décembre 2015

Sommaire

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.cpp
Problème :

- Pour utiliser un std : :function<> J'ai besoin du type de retour du foncteur et du type du paramètre.
- J'ai également besoin du type du paramètre pour d'autre chose (l'operator : :* par exemple)

Solution :

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

Sommaire

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 ?