STD::ENABLE_IF

UTILISATION DU SFINAE POUR ACTIVER/DÉSACTIVER DES OVERLOADS À LA COMPILATION

Michaël ROYNARD - fxcash-dev

SFINAE (DÉFINITION)

SUBSTITUTION FAILURE IS NOT AN ERROR

Lorsque la substitution d'un type déduit d'un paramètre template échoue cela n'est pas considéré comme une erreur. A la place, la spécialisation est juste écartée de la liste des overloads disponibles pour l'ADL (Argument dependent lookup).

PROTOTYPE & IMPLEM STD::ENABLE_IF

```
template<bool B, typename T = void>
struct enable_if;
```

```
template<bool B, typename T = void>
struct enable_if {};

// Utilisation du SFINAE pour spécialisation à true
template<typename T>
struct enable_if<true, T> { using type = T; };

// Laisse le compilateur échouer la compilation lors d'un accè
// enable_if<false, T>::type car type ne sera pas défini

template<bool B, typename T = void>
using enable_if_t = typename enable_if<B, T>::type; // C++14
```

UTILISATIONS

5 EMPLACEMENT POSSIBLES

```
template<
  typename T, typename U = T,
  /*(1)*/enable_if_t<is_same<T, int>::value, int> = 0,
  /*(2)*/typename = enable_if_t<is_same<T, U>::value, T>
  >
/*(3)*/enable_if_t<is_integral<U>::value, int>
f(T a,
  /*(4)*/enable_if_t<is_same<U, int>::value, U> b,
  /*(5)*/enable_if_t<is_same<T, int>::value, void**> = nullptr
{
  cout << "void f(T a, U a) when T = U = int" << endl;
  return a;
}
```

```
f(1, 1);  // affiche "void f(T a, U a) when T = U = int"
f(1., 1.);// error: no matching function for call to
          //        'f(double, double)'
```

EMPLACEMENTS

1. valeur template par défaut
2. template typename
3. type de retour
4. paramètre de fonction
5. paramètre de fonction additionnel

(1) VALEUR TEMPLATE PAR DÉFAUT

"EMULE" LES CONCEPTS

```
template<
  typename ... Args,
  enable_if_t<less<size_t>{}(sizeof...(Args), 1), // x < 1
              int> = 0
  >
void f(Args&& ... args) {
  cout << "Empty pack !" << endl;
}
```

```
template<
  typename ... Args,
  enable_if_t<less<size_t>{}(sizeof...(Args), 3), // x < 3
              int> = 0
  >
void f(Args&& ... args) {
  cout << "Optimized for pack size < 3 !" << endl;
}
```

```
#define REQUIRES(COND) std::enable_if_t<(COND), int> = 0
```

(1) VALEUR TEMPLATE PAR DÉFAUT

ATTENTION AUX CONDITIONS

```
template<
  typename ... Args,
  enable_if_t<greater<size_t>{}(sizeof...(Args), 2), // x >= 3
              int> = 0
  >
void f(Args&& ... args) {
  cout << "Generic data processing !" << endl;
}
```

```
f(0);   // affiche "Optimized for pack size < 3 !"
f();    // error: call of overloaded 'f(void)' is ambiguous
```

(1) VALEUR TEMPLATE PAR DÉFAUT

REQUIERT DES CONDITIONS D'EXCLUSION MUTUELLES SCRICTES

```
template<
  typename ... Args,
  enable_if_t<less<size_t>{}(sizeof...(Args), 3) && // x < 3
              greater<size_t>{}(sizeof...(Args), 0), // x >= 1
              int> = 0
  >
void f(Args&& ... args) {
  cout << "Optimized for pack size < 3 !" << endl;
}
```

```
f();                // Empty pack !
f(0);               // "Optimized for pack size < 3 !"
f("test", 2.);      // Generic data processing !
f(1, "test", 3.);// Generic data processing !
```

UTILISATIONS

5 EMPLACEMENTS POSSIBLES

```
template<
  typename T, typename U = T,
  /*(1)*/enable_if_t<is_same<T, int>::value, int> = 0,
  /*(2)*/typename = enable_if_t<is_same<T, U>::value, T>
  >
/*(3)*/enable_if_t<is_integral<U>::value, int>
f(T a,
  /*(4)*/enable_if_t<is_same<U, int>::value, U> b,
  /*(5)*/enable_if_t<is_same<T, int>::value, void**> = nullptr
{
  cout << "void f(T a, U a) when T = U = int" << endl;
  return a;
}
```

	Valeur template par défaut (1)	Template typename (2)
Variadic Pack	OK	OK
Multiple overload	OK	KO
Spécialisation de template	KO	OK
ctors / dtors / conv. operators	OK	OK

(3) TYPE DE RETOUR

PERMET D'ÉMULER L'OVERLOAD PAR TYPE DE RETOUR

```
template<bool>
struct static_if {
  template<class Ret, class If, class Else, class ... Args>
  static Ret call(If&&, Else&&, Args&& ...);
};
```

```
constexpr bool condition = true;
auto ret = static_if<condition>::call<double>(
    [](int a, double b) { return b * a; },  // if lambda
    [](int a, double b) { 0.; },            // else lambda
    15, 42.                                 // arguments
);
cout << ret << endl; // affiche 630
```

(3) TYPE DE RETOUR

PROBLÈME À L'IMPLÉMENTATION

```
template<>
struct static_if<true> {
  template<class Ret, class If, class Else, class ... Args>
  Ret call(If&& if_f, Else&&, Args&& ... args)
  {
    return if_f(forward<Args>(args)...);
  }
};
```

```
template<>
struct static_if<true> {
  template<class Ret, class If, class Else, class ... Args>
  static Ret call(If&&, Else&& else_f, Args&& ... args)
  {
    return else_f(forward<Args>(args)...);
  }
};
```

(3) TYPE DE RETOUR

SOLUTION : ENABLE_IF (IMPLEM TRUE)

```
template<>
struct static_if<true> {
  template<class Ret, class If, class Else, class ... Args>
  static auto call(If&& if_f, Else&&, Args&& ... args)
    -> enable_if_t<!is_same<Ret, void>::value, Ret>
  {
    return if_f(forward<Args>(args)...);
  }

  template<class Ret, class If, class Else, class ... Args>
  static auto call(If&& if_f, Else&&, Args&& ... args)
    -> enable_if_t<is_same<Ret, void>::value>
  {
    if_f(forward<Args>(args)...);
  }
};
```

(3) TYPE DE RETOUR

SOLUTION : ENABLE_IF (IMPLEM FALSE)

```
template<>
struct static_if<false> {
  template<class Ret, class If, class Else, class ... Args>
  static auto call(If&&, Else&& else_f, Args&& ... args)
    -> enable_if_t<!is_same<Ret, void>::value, Ret>
  {
    return else_f(forward<Args>(args)...);
  }

  template<class Ret, class If, class Else, class ... Args>
  static auto call(If&&, Else&& else_f, Args&& ... args)
    -> enable_if_t<is_same<Ret, void>::value>
  {
    else_f(forward<Args>(args)...);
  }
};
```

UTILISATIONS

5 EMPLACEMENTS POSSIBLES

```
template<
  typename T, typename U = T,
  /*(1)*/enable_if_t<is_same<T, int>::value, int> = 0,
  /*(2)*/typename = enable_if_t<is_same<T, U>::value, T>
  >
/*(3)*/enable_if_t<is_integral<U>::value, int>
f(T a,
  /*(4)*/enable_if_t<is_same<U, int>::value, U> b,
  /*(5)*/enable_if_t<is_same<T, int>::value, void**> = nullptr
{
  cout << "void f(T a, U a) when T = U = int" << endl;
  return a;
}
```

	Valeur template par défaut (1)	Template typename (2)	Type de retour (3)
Variadic Pack	OK	OK	OK
Multiple overload	OK	KO	OK
Spécialisation de template	KO	OK	OK
ctors / dtors / conv. operators	OK	OK	KO

(4) PARAMÈTRE DE FONCTION

```
template<typename T>
void f(T a,
       enable_if_t<is_integral<T>::value, T> b)
{
  cout << "integral type !" << endl;
}
```

```
template<typename T>
void f(T a,
       enable_if_t<!is_integral<T>::value, T> b)
{
  cout << "not integral type !" << endl;
}
```

```
f(0, 0);     // affiche "integral type !"
f("t", "t");// affiche "not integral type!"
```

(4) PARAMÈTRE DE FONCTION

ATTENTION : LA DÉDUCTION N'EST PAS AUTOMATIQUE !

```
template<typename T>
void f(enable_if_t<is_integral<T>::value, T> a) {
  cout << "integral type !" << endl;
}
```

```
template<typename T>
void f(enable_if_t<!is_integral<T>::value, T> a) {
  cout << "not integral type !" << endl;
}
```

```
f(0);  //error: no matching function for call to 'f(int)'
f("t");//error: no matching function for call to 'f(const char
f<int>(0);                  // affiche "integral type !"
f<const char(&)[2]>("t"); // affiche "not integral type!"
```

(5) PARAMÈTRE DE FONCTION

S'EN SORTIR AVEC LE PARAMÈTRE ADDITIONNEL

```
template<typename T>
void f(T a,
       enable_if_t<is_integral<T>::value, void**> = nullptr)
{
  cout << "integral type !" << endl;
}
```

```
template<typename T>
void f(T a,
       enable_if_t<!is_integral<T>::value, void**> = nullptr)
{
  cout << "not integral type !" << endl;
}
```

```
f(0);  // affiche "integral type !"
f("t");// affiche "not integral type!"
```

UTILISATIONS

5 EMPLACEMENTS POSSIBLES

```
template<
  typename T, typename U = T,
  /*(1)*/enable_if_t<is_same<T, int>::value, int> = 0,
  /*(2)*/typename = enable_if_t<is_same<T, U>::value, T>
  >
/*(3)*/enable_if_t<is_integral<U>::value, int>
f(T a,
  /*(4)*/enable_if_t<is_same<U, int>::value, U> b,
  /*(5)*/enable_if_t<is_same<T, int>::value, void**> = nullptr
{
  cout << "void f(T a, U a) when T = U = int" << endl;
  return a;
}
```

	Valeur template par défaut (1)	Template typename (2)	Type de retour (3)	Param. fonction (4)	Param. fonctio addition (5)
Variadic Pack	OK	OK	OK	OK*	KO
Multiple overload	OK	KO	OK	OK	OK
Spécialisation de template	KO	OK	OK	OK	OK
ctors / dtors / conv. operators	OK	OK	KO	OK	OK

* : le paramètre porteur du enable_if doit obligatoirement être non variadic

RESSOURCES

- cppreference : SFINAE
- cppreference : std::enable_if
- CppCon 2016: Stephan T. Lavavej “tuple＜＞: What's New and How it Works"
- Slides on github