# C++ Template Specialization Using Enable If

06-16-2022 06-16-2022 BLOG 7 MINUTES READ (ABOUT 10

## Introduction

In C++ metaprogramming, `std::enable_if` is an important function to enable certain types for template specialization via some predicates known at the compile time. Using types that are not enabled by `std::enable_if` for template specialization will result in compile-time error.

In this blog post, I would like to discuss how to understand C++ `std::enable_if` with an emphasis on its application in template parameters.

## Non-Type and Type Template Parameters

To understand `std::enable_if`, it is necessary to understand the non-type and type template parameters. Let's get ourselves familiar with them by looking at two examples.

### Non-Type Template Parameters

Please pay special attention to the class `C` as this usage is often used with `std::enable_if`.

non_type_template_parameter.cpp

```cpp
template <int N>
class A
{
    int v{N};
};

template <int N = 10>
class B
{
    int v{N};
};

template <int = 10>
```

```
class C
{
    int v{0};
};

template <int>
class D
{
    int v{0};
};

template <int, int>
class E
{
    int v{0};
};

int main()
{
    A<10> a{};
    B<> b{};
    C<> c{};
    D<10> d{};
    E<10, 10> e{};
}
```

### Type Template Parameters

Please pay special attention to the class `C` as this usage is often used with `std::enable_if`.

type_template_parameter.cpp

```
template <typename T>
class A
{
    T v{0};
};

template <typename T = int>
class B
{
    T v{0};
};

template <typename = int>
class C
{
    int v{0};
};
```

```cpp

template <typename>
class D
{
    int v{0};
};


template <typename, typename>
class E
{
    int v{0};
};

int main()
{
    A<int> a{};
    B<> b{};
    C<> c{};
    D<int> d{};
    E<int, int> e{};
}
```

## Template Specialization Using Enable If

### std::enable_if

In C++, the class signature of `std::enable_if` is as follows.

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

If `B` is true, `std::enable_if` has a public member typedef `type`, equal to `T`; otherwise, there is no member typedef.

`std::enable_if` could be implemented as follows.

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

template<class T>
struct enable_if<true, T> { typedef T type; };
```

This means, whenever the implementation tries to access `enable_if<B,T>::type` when `B = false`, the compiler will raise compilation error, as the object member `type` is not defined.

Since C++14, there is an additional helper shortcut `std::enable_if_t` defined in the C++ standard library.

```cpp
template< bool B, class T = void >
using enable_if_t = typename enable_if<B,T>::t
```

### Enable Template Specialization Via Template Parameters

`std::enable_if` or `std::enable_if_t` could be used for restricting or enabling the types used for template specialization via template parameters. Any undesired types used for template specialization will be prevented by compiler.

Let's check an example of enabling only one type or types for a template function. Here we enabled integer types for the function `foo` and `bar` using the predicate `std::is_integral<T>::value`.

```cpp
#include <iostream>
#include <type_traits>

template <typename T, typename = std::enable_


void foo()
{
    std::cout << "T could only be int" << std
}

template <typename T,
          std::enable_if_t<std::is_integral<T
              true> // It does not matter wha
                    // as long as the value i
void bar()
{
    std::cout << "T could only be int" << std
}

int main()
{
```

```
    foo<int>();
    // foo<float>(); // Compilation error.
    bar<int>();
    // bar<float>(); // Compilation error.
}
```

Notice that in this case we used the type template parameter compile-time checking for the function `foo` and used the non-type template parameter compile-time checking for the function `bar`.

When it comes to enabling multiple types for a template function, the type template parameter compile-time checking will be prevented by compiler as the declarations are treated as redeclarations of the same function template.

Here we enabled both integer types and floating point types for the function `bar` using the predicate `std::is_integral<T>::value` and `std::is_floating_point<T>::value`, respectively.

```
#include <iostream>
#include <type_traits>

template <typename T, typename = std::enable_


void foo()
{
    std::cout << "T is int" << std::endl;
}

// Compile-time error: redefinition
// template <typename T,
//           typename = std::enable_if_t<std:
//           float>>
// void foo()
// {
//     std::cout << "T is float" << std::endl
// }

template <typename T,
          std::enable_if_t<std::is_integral<T
              true> // It does not matter whc
                    // as long as the value i
void bar()
```

```
{
    std::cout << "T is int" << std::endl;
}

template <typename T,
          std::enable_if_t<std::is_floating_p
              true> // It does not matter whc
                    // as long as the value i
void bar()
{
    std::cout << "T is float" << std::endl;
}

int main()
{
    foo<int>();
    // foo<float>();
    bar<int>();
    bar<float>();
}
```

The `std::is_integral<T>::value` and `std::is_floating_point<T>::value` are mutually exclusive and only one can be `true` at compile time for one specialization.

As we have discussed previously, the program below is equivalent as the program above and it could be compiled with C++11 standard.

```
#include <iostream>
#include <type_traits>

template <typename T,
          typename = typename std::enable_if<


void foo()
{
    std::cout << "T is int" << std::endl;
}

// Compile-time error: redefinition
// template <typename T,
//           typename = std::enable_if_t<std:
//           float>>
// void foo()
// {
```

```
//     std::cout << "T is float" << std::endl
// }

template <typename T,
          typename std::enable_if<std::is_int
              true> // It does not matter whc
                    // as long as the value i
void bar()
{
    std::cout << "T is int" << std::endl;
}

template <
    typename T,
    typename std::enable_if<std::is_floating_
        true> // It does not matter what type
              // as long as the value is of t
void bar()
{
    std::cout << "T is float" << std::endl;
}

int main()
{
    foo<int>();
    // foo<float>();
    bar<int>();
    bar<float>();
}
```

To understand `std::enable_if`, for example, when `std::is_integral<T>::value` is evaluated to be `true` at compile time, `std::enable_if_t<std::is_integral<T>::value, bool> = true` is equivalent `std::enable_if_t<true, bool> = true` and is equivalent as `typename std::enable_if<true, bool>::type = true` and is equivalent as `typename bool = true`. The remaining `typename` is probably an indicator and will be removed during preprocessing. So ultimately what compiler will see is `bool = true` which is exactly the same as the class `C` scenario in the non-type template parameters.

The same analysis could be performed on `typename = std::enable_if_t<std::is_integral<T>::value, float>` as well to help the understanding.

## Enable Template Specialization Via Others

std::enable_if or std::enable_if_t could be used for restricting or enabling the types used for template specialization via return type or function parameters. Understanding those is almost equivalent as understand enabling template specialization via template parameters, and I am not going to elaborate it here.

## References


- Template Parameters and Template Arguments
- std::enable_if - CPP Reference


C++ Template Specialization Using Enable If
https://leimao.github.io/blog/CPP-Enable-If/

| Author | Posted on | Updated on | Licensed under |
| --- | --- | --- | --- |
| Lei Mao | 06-16-2022 | 06-16-2022 | |

CPP




< Sunol Wilderness Regional Preserve 徒步   圣伯纳犬和酒桶 >


## Comments