Controlling overload resolution with SFINAE

MUC++ Lightning Talks
2017-09-13

Short follow-up on the discussion during Roland Bock's presentation "Template Magic for Beginners" on 2017-08-31

Matthäus Brandl
brandl.matthaeus@gmail.com

Controlling overload resolution with SFINAE

Question:

Is it possible to have more than two paths to choose from when controlling overload resolution with SFINAE?

Used type traits

- `std::is_pointer`
- `std::is_pod`

  Note that both traits are independent - a type can be both, only POD or neither

- `std::is_integral`
- `std::is_floating_point`
- `std::is_arithmetic`

  Note that `is_arithmetic` is the union of `is_integral` and `is_floating_point`.
  Using its negation will result in disjoint traits.

Note on SFINAE syntax

Roland Bock used the following syntax in his presentation:

```
template <class T, typename = std::enable_if_t<std::is_pointer<T>::value>>
void foo(T)
{
    // ...
}
```

The syntax surprised me at first, but then I realized that it is a really short and elegant way of using `std::enable_if`.

However this does not fare well if used for more than one overload. Turning one overload off is possible, but selecting from several overloads fails. Your code would not compile, even if you do not instantiate any of the function templates.

cppreference explains why this is illegal:

> "... because default template arguments are not part of [the] function template's signature, and declaring two different function templates with the same signature is illegal."

Oppositional conditions work

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

template <class T, std::enable_if_t<std::is_pointer<T>::value, int> = 0>
void foo(T)
{
    std::cout << "foo<is_pointer>()" << std::endl;
}

template <class T, std::enable_if_t<!std::is_pointer<T>::value, int> = 0>
void foo(T)
{
    std::cout << "foo<!is_pointer>()" << std::endl;
}

int main()
{
    foo(5);
    foo(&foo<int>);
}
```

```
foo<!is_pointer>()
foo<is_pointer>()
```

Live on Coliru

Disjoint conditions work

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

template <class T, std::enable_if_t<std::is_integral<T>::value, int> = 0>
void foo(T)
{
    std::cout << "foo<is_integral>()" << std::endl;
}

template <class T, std::enable_if_t<std::is_floating_point<T>::value, int> = 0>
void foo(T)
{
    std::cout << "foo<is_floating_point>()" << std::endl;
}

template <class T, std::enable_if_t<!std::is_arithmetic<T>::value, int> = 0>
void foo(T)
{
    std::cout << "foo<!std::is_arithmetic>()" << std::endl;
}

int main()
{
    foo(42);
    foo(3.14);
    foo(&foo<int>);
}
```

```
foo<is_integral>()
foo<is_floating_point>()
foo<!std::is_arithmetic>()
```

Live on Coliru

Overlapping conditions fail

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

template <class T, std::enable_if_t<std::is_pointer<T>::value, int> = 0>
void foo(T)
{
    std::cout << "foo<is_pointer>()" << std::endl;
}

template <class T, std::enable_if_t<std::is_pod<T>::value, int> = 0>
void foo(T)
{
    std::cout << "foo<is_pod>()" << std::endl;
}

int main()
{
    foo(static_cast<int*>(nullptr));
}
```

Live on Coliru

```
main.cpp: In function 'int main()':
main.cpp:18:35: error: call of overloaded 'foo(int*)' is ambiguous
     foo(static_cast<int*>(nullptr));
                                   ^
main.cpp:5:6: note: candidate: void foo(T) [with T = int*;
              typename std::enable_if<std::is_pointer<_Tp>::value, int>::type <anonymous> = 0]
 void foo(T)
      ^~~
main.cpp:11:6: note: candidate: void foo(T) [with T = int*;
               typename std::enable_if<std::is_pod<_Tp>::value, int>::type <anonymous> = 0]
 void foo(T)
      ^~~
```

Or do they?

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

template <class T, std::enable_if_t<std::is_pointer<T>::value, int> = 0>
void foo(T)
{
    std::cout << "foo<is_pointer>()" << std::endl;
}

template <class T, std::enable_if_t<std::is_pod<T>::value, int> = 0>
void foo(T)
{
    std::cout << "foo<is_pod>()" << std::endl;
}

int main()
{
    foo(nullptr);
}
```

```
foo<is_pod>()
```

Live on Coliru

Controlling overload resolution with SFINAE

Thanks for your attention