3-sfinae

June 3, 2024

1 At the crossroads between overload and templates

1.1 Motivation

When overloaded function templates and ordinary functions are competing, the function templates are often invasive and the results may be surprising.

Below, what do you think will be displayed ? `(template) 42`, because `42` is by default of type `int`, and will perfectly match the function template, while the ordinary function requires a conversion. We would have prefered that the function template is only used with floating point types !

[9]:
```
%%file tmp.sfinae.cpp

#include <iostream>

bool equal( unsigned e1, unsigned e2 )
 {
  std::cout<<"(unsigned)"<<std::endl ;
  return (e1==e2) ;
 }

template< typename T >
bool equal( T e1, T e2 )
 {
  std::cout<<"(template)"<<std::endl ;
  return abs(e1-e2)<1e-6 ;
 }

int main()
 { equal(100,10*10) ; }
```

```
Overwriting tmp.sfinae.cpp
```

[10]:
```
!rm -f tmp.sfinae.exe && g++ -std=c++17 tmp.sfinae.cpp -o tmp.sfinae.exe
```

[11]:
```
!./tmp.sfinae.exe
```

```
(template)
```

1.2 Overload resolution

When the compiler encounters a function call, it looks for all candidate functions with the appropriate name, and then selects the one that best matches the arguments of the call: this is what we call the "overload resolution".

When it examines a candidate function that is a template, the compiler substitutes the types if they are explicitly specified in the call, or it tries to infer them from the function's call arguments.

Let's change the template return type :

[26]:
```
%%file tmp.sfinae.cpp

#include <iostream>
#include <limits>

bool equal( unsigned e1, unsigned e2 )
 {
  std::cout<<"(unsigned)"<<std::endl ;
  return (e1==e2) ;
 }

template< typename T >
typename T::bool_type equal( T e1, T e2 )
 {
  std::cout<<"(template)"<<std::endl ;
  return abs(e1-e2)<std::numeric_limits<T>::epsilon() ;
 }

int main()
 { equal(100,10*10) ; }
```

```
Overwriting tmp.sfinae.cpp
```

[27]:
```
!rm -f tmp.sfinae.exe && g++ -std=c++17 tmp.sfinae.cpp -o tmp.sfinae.exe
```

[28]:
```
!./tmp.sfinae.exe
```

```
(unsigned)
```

This time, the first variant is selected. But all candidates were reviewed, and the template was interpreted (and rejected) in this format :

```
int::bool_type equal( int e1, int e2 )
```

This is an invalid form, since the predefined type `int` has no nested `bool_type`. One could imagine that the compiler emits an error in this case, but this would make the generic code writing extremely difficult. The standard therefore recommends that when the type deduction has failed, then the candidate template should be ignored, but **it is not a compilation error**.

1.3 SFINAE

In C++ jargon, this rule is called SFINAE (Substitution Failure Is Not An Error). The standard specifies that the rule applies to the the *immediate environment* of the function template, i.e. everything that is involved in the declaration of the function, not in its definition, i.e. the body of the function. Let's modify a little our previous example:

[15]:
```
%%file tmp.sfinae.cpp

#include <iostream>

bool equal( unsigned e1, unsigned e2 )
 {
  std::cout<<"(unsigned)"<<std::endl ;
  return (e1==e2) ;
 }

template< typename T >
bool equal( T e1, T e2 )
 {
  std::cout<<"(template)"<<std::endl ;
  typename T::bool_type res = abs(e1-e2)<1e-6 ;
  return res ;
 }

int main()
 { equal(100,10*10) ; }
```

```
Overwriting tmp.sfinae.cpp
```

[16]:
```
!rm -f tmp.sfinae.exe && g++ -std=c++17 tmp.sfinae.cpp -o tmp.sfinae.exe
```

```
tmp.sfinae.cpp: In instantiation of 'bool equal(T, T) [with T = int]':
tmp.sfinae.cpp:19:19:   required from here
tmp.sfinae.cpp:14:25: error: 'int' is not a class, struct, or union type
   14 |   typename T::bool_type res = abs(e1-e2)<1e-6 ;
      |                         ^~~
tmp.sfinae.cpp:14:25: error: 'int' is not a class, struct, or union type
```

This time, if the candidate function is selected to solve a function call, with a builtin type for `T` (e.g. `int` as before), we will indeed have a compilation error. Of course, we prefer when the candidate function is only ignored, and does not hide another function which would fit better.

The moral is that when you write a function template that has no mean with certain types, **try to formulate a declaration that will cause the type substitution to fail**, so that an inappropriate type does not go beyond the stage of the overload resolution.

1.4 enable_if

Very early in the history of C++, SFINAE became a mechanism that was used on purpose, when people realized it allowed to write templates triggered only by appropriate parameters, instead of catching everything that comes along.

`enable_if` is a very appreciated utility meta-fonction: it is receiving an input type and a boolean expression, and it "returns" the same type as output, if and only if the boolean is true. Of course, C++14 brings a `_t` alias:

[1]:
```
template <bool, typename T = void>
struct enable_if
 {} ;

template <typename T>
struct enable_if<true, T>
 { using type = T ; } ;

template <bool b, typename T>
using enable_if_t = typename enable_if<b, T>::type ;
```

In the context of a given template, by replacing any occurrence of a type `T` with an `enable_if_t<condition,T>`, we make sure that the substitution fails if the condition is `false`, and that the template is discarded when solving the overload. For example:

[1]:
```
#include <iostream>
```

[2]:
```
template <typename T>
bool equal( T e1, T e2, std::enable_if_t<std::is_floating_point_v<T>,T> epsilon␣
 ↪= 1e-6 )
 {
  std::cout<<"(template)"<<std::endl ;
  return abs(e1-e2)<epsilon ;
 }
```

In order not to affect readability too much, it is now recommended to add a fake template parameter, i.e. a pointer to an `enable_if`, with a default value of `nullptr`. This parameter is of no use, except to make the resolution of the overload fail when the condition is `false`.

[3]:
```
%%file tmp.sfinae.cpp

#include <iostream>
#include <type_traits>

bool equal( unsigned e1, unsigned e2 )
 {
  std::cout<<"(unsigned)"<<std::endl ;
  return (e1==e2) ;
 }
```

```
template< typename T, std::enable_if_t<std::is_floating_point_v<T>> * = nullptr␣
 ↪>
bool equal( T e1, T e2, T epsilon = 1e-6 )
 {
  std::cout<<"(template)"<<std::endl ;
  return abs(e1-e2)<epsilon ;
 }

int main()
 { equal(100,10*10) ; }
```

```
Writing tmp.sfinae.cpp
```

[4]:
```
!rm -f tmp.sfinae.exe && g++ -std=c++17 tmp.sfinae.cpp -o tmp.sfinae.exe
```

[5]:
```
!./tmp.sfinae.exe
```

```
(unsigned)
```

At this stage, we can even put in competition two function templates, one that activates for floating numbers, and the other for non-floating numbers.

[6]:
```
%%file tmp.sfinae.cpp

#include <iostream>
#include <type_traits>

template< typename T, std::enable_if_t<!std::is_floating_point_v<T>> * =␣
 ↪nullptr >
bool equal( T e1, T e2 )
 { std::cout<<"(non floating)"<<std::endl ; return (e1==e2) ; }

template< typename T, std::enable_if_t<std::is_floating_point_v<T>> * = nullptr␣
 ↪>
bool equal( T e1, T e2, T epsilon = 1e-6 )
 { std::cout<<"(floating)"<<std::endl ; return abs(e1-e2)<epsilon ; }

int main()
 {
  equal(100,10*10) ;
  equal(1.,.1+.1+.1+.1+.1+.1+.1+.1+.1+.1) ;
 }
```

```
Overwriting tmp.sfinae.cpp
```

[7]:
```
!rm -f tmp.sfinae.exe && g++ -std=c++17 tmp.sfinae.cpp -o tmp.sfinae.exe
```

[8]:
```
!./tmp.sfinae.exe
```

```
(non floating)
(floating)
```

2 Questions ?

3 Exercise

1. Instead of the `static_assert`, use an `std::enable_if<>` in the context of the `times_power_of_two()` function. Check that the compiler always refuses to compile `times_power_of_two(3.14,1)`, on the grounds that it can't find a `times_power_of_two` for the type of 3.14.
2. Uncomment the second function `times_power_of_two()` for doubles, and transform it into a template similar to the above one, yet checking the type parameter is a floating point type.
3. Finally, try to join them into a single universal `times_power_of_two()` function, using `if constexpr`.

[ ]:
```
%%file tmp.sfinae.cpp

#include <iostream>

template < typename T >
T times_power_of_two( T number, int exponent )
 {
  static_assert(std::is_integral_v<T>,"the type must be an integer !") ;
  if (exponent<0) { return (number>>-exponent) ; }
  else { return (number<<exponent) ; }
 }

//double times_power_of_two( double number, int exponent )
// {
//  while (exponent<0) { number /= 2 ; exponent++ ; }
//  while (exponent>0) { number *= 2 ; exponent-- ; }
//  return number ;
// }

int main()
{
  std::cout<<times_power_of_two(42,1)<<std::endl ;
  std::cout<<times_power_of_two(42,-1)<<std::endl ;
  std::cout<<times_power_of_two(3.14,1)<<std::endl ;
  std::cout<<times_power_of_two(3.14,-1)<<std::endl ;
  return 0 ;
}
```

[ ]:
```
!rm -f tmp.sfinae.exe && g++ -std=c++17 tmp.sfinae.cpp -o tmp.sfinae.exe
```

[ ]:
```
!./tmp.sfinae.exe
```