2-types-handling

June 3, 2024

1 Types handling

In the context of C++, we will call ***metafunction*** a function that is **executed by the compiler**, at compile time. Its inputs are the parameters of a template, and can therefore be **types and integer constant expressions**. Similarly, the outputs can be types or integers. The gymnastics required to access the results depends greatly on the version of C++ used.

1.1 Example of type transformation

Below, the `remove_possible_pointer` metafunction removes the possible higher-level pointer. In this old C++ implementation, the input type is U, passed as a template parameter, and the result of the meta-function is extracted using the syntax `typename remove_possible_pointer<U>::type`:

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

[3]:
```
template<typename U>    // general case
struct remove_possible_pointer
 { typedef U type ; } ;

template<typename T>    // pour U = T*
struct remove_possible_pointer<T*>
 { typedef T type ; } ;
```

[4]:
```
template<typename U>
char const * value_type_name( U /* variable */ )
 { return typeid(typename remove_possible_pointer<U>::type).name() ; }
```

[5]:
```
std::cout<<value_type_name(42)<<std::endl ;
std::cout<<value_type_name(new double(3.14))<<std::endl ;
```

```
i
d
```

C++11 offers a new comfort with template aliases. We can add an alias to the previous code, to make it easier to use:

[6]:
```
template<typename T>
using remove_possible_pointer_t = typename remove_possible_pointer<T>::type ;
```

[7]:
```
template<typename U>
char const * value_type_name( U /* variable */ )
 { return typeid(remove_possible_pointer_t<U>).name() ; }
```

[8]:
```
std::cout<<value_type_name(42)<<std::endl ;
std::cout<<value_type_name(new double(3.14))<<std::endl ;
```

```
i
d
```

1.2 New standard library

C++11 introduces many meta-fonctions for the transformation of types. They are all implemented individually to facilitate their composition, rather than grouped together, like in the C++03 library. We list the easiest ones below. Since C++14, for each of them, there is an additional alias template `_t`, like in the previous example.

[9]:
```
namespace std
 {
  // const-volatile modifications:
  template <class T> struct remove_const;
  template <class T> struct remove_volatile;
  template <class T> struct remove_cv;
  template <class T> struct add_const;
  template <class T> struct add_volatile;
  template <class T> struct add_cv;

  // reference modifications:
  template <class T> struct remove_reference;
  template <class T> struct add_lvalue_reference;
  template <class T> struct add_rvalue_reference;

  // sign modifications:
  template <class T> struct make_signed;
  template <class T> struct make_unsigned;

  // array modifications:
  template <class T> struct remove_extent;
  template <class T> struct remove_all_extents;

  // pointer modifications:
  template <class T> struct remove_pointer;
  template <class T> struct add_pointer;
 }
```

1.3 Example of type testing

Below, the `is_floating_point` metafunction is used to determine if the input type is a floating number. This time, the result of the meta-function is a boolean (from the compiler's point of view, a kind of integer), and is extracted using the syntax `is_floating_point<U>::value`:

[10]:
```
template <typename T>
struct is_floating_point
 { static const bool value = false ; } ;

template <>
struct is_floating_point<float>
 { static const bool value = true ; } ;

template <>
struct is_floating_point<double>
 { static const bool value = true ; } ;
```

[11]:
```
#include <iostream>
```

[12]:
```
template< typename T>
void check( T v )
{
  if (is_floating_point<T>::value)
    std::cout << v << " is a floating point number" << std::endl ;
  else
    std::cout << v << " is not a floating point number" << std::endl ;
}
```

[13]:
```
check(42) ;
check(3.14) ;
```

```
42 is not a floating point number
3.14 is a floating point number
```

C++14 offers a little more comfort with variable aliases. We can add an alias to the previous code to make it easier to use, saving us from typing `::value`:

[20]:
```
template <typename T>
bool is_floating_point_v = is_floating_point<T>::value ;
```

[21]:
```
template< typename T>
void check( T v )
{
  if (is_floating_point_v<T>)
    std::cout << v << " is a floating point number" << std::endl ;
  else
    std::cout << v << " is not a floating point number" << std::endl ;
}
```

[22]:
```
check(42) ;
check(3.14) ;
```

```
42 is not a floating point number
3.14 is a floating point number
```

1.4 In the standard library

C++11 introduces a bunch of new traits, e.g. `is_fundamental<T>` (is T a builtin type ?), `is_array<T>` (is T an array?), and `is_base_of<T1, T2>` (is T2 a base class of T1, or T1 itself?). Note that these new traits are all implemented individually to facilitate their composition, rather than grouped together like the C++03 library.

[23]:
```
// primary type categories:
template <class T> struct is_void;
template <class T> struct is_null_pointer;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_lvalue_reference;
template <class T> struct is_rvalue_reference;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;
```

[24]:
```
// composite type categories:
template <class T> struct is_reference;
template <class T> struct is_arithmetic;
template <class T> struct is_fundamental;
template <class T> struct is_object;
template <class T> struct is_scalar;
template <class T> struct is_compound;
template <class T> struct is_member_pointer;
```

[25]:
```
// type properties:
template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_trivial;
template <class T> struct is_trivially_copyable;
template <class T> struct is_standard_layout;
template <class T> struct is_pod;
template <class T> struct is_literal_type;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct is_signed;
template <class T> struct is_unsigned;
```

```
template <class T, class... Args> struct is_constructible;
template <class T> struct is_default_constructible;
template <class T> struct is_copy_constructible;
template <class T> struct is_move_constructible;
template <class T, class U> struct is_assignable;
template <class T> struct is_copy_assignable;
template <class T> struct is_move_assignable;
template <class T> struct is_destructible;
//...
```

[26]:
```
// type relations:
template <class T, class U> struct is_same;
template <class Base, class Derived> struct is_base_of;
template <class From, class To> struct is_convertible;
```

1.5 C++17 flavor

Starting with C++17, all the above `is_something` templates, whose result is a variable (usually boolean), and which is accessed by `is_something<T>::value`, have a corresponding variable template `is_something_v<T>`, to save the typing of `::value`. Also, since those values can be evaluated at compile time, they can be combined with the new `if constexpr`.

[34]:
```
%%file tmp.types-handling.cpp

#include <iostream>

template <typename T>
struct is_floating_point
 { static const bool value = false ; } ;

template <>
struct is_floating_point<float>
 { static const bool value = true ; } ;

template <>
struct is_floating_point<double>
 { static const bool value = true ; } ;

template <typename T>
constexpr bool is_floating_point_v = is_floating_point<T>::value ;

template< typename T>
void check( T v )
{
  if constexpr (is_floating_point_v<T>)
    std::cout << v << " is a floating point number" << std::endl ;
  else
```

```
    std::cout << v << " is not a floating point number" << std::endl ;
}

int main() {
  check(42) ;
  check(3.14) ;
}
```

```
Overwriting tmp.types-handling.cpp
```

[35]:
```
!rm -f tmp.types-handling.exe && g++ -std=c++17 tmp.types-handling.cpp -o tmp.
 ↪types-handling.exe
```

[36]:
```
!./tmp.types-handling.exe
```

```
42 is not a floating point number
3.14 is a floating point number
```

1.6 What for ?

- In `static_assert`, to check template arguments.
- In `if constexpr`, to select a specialized implementation.
- In the so-called SFINAE mechanism…

2 Questions ?

3 Exercice

1. Make the `times_power_of_two()` function applicable to any type. Notice that the compilation fails for `times_power_of_two(3.14,1)`, because the offset operators `<<` and `>>` do not exist for the `double` type.
2. Make `times_power_of_two()` only apply to integer types, using `static_assert` and `std::is_integral<>`. Check that now the compiler refuses to compile `times_power_of_two(3.14,1)`, because the inferred type is not integer.

[37]:
```
%%file tmp.types-handling.cpp

#include <iostream>

int times_power_of_two( int number, int exponent )
 {
  if (exponent<0) { return (number>>-exponent) ; }
  else { return (number<<exponent) ; }
 }

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 ;
}
```

```
Overwriting tmp.types-handling.cpp
```

[38]:
```
!rm -f tmp.types-handling.exe && g++ -std=c++17 tmp.types-handling.cpp -o tmp.
 ↪types-handling.exe
```

[39]:
```
!./tmp.types-handling.exe
```

```
84
21
6
1
```