Лекция 2.75. Все те же примеры метапрограммирования

Метапрограммирование в C++

25 октября 2018 г.

Частичный порядок соответствия шаблонов

При наличии нескольких перегруженных версий шаблона нужно выбрать между ними.

Пример

Из стандарта: (1.2). Если найдено более одной соответствующей специализации, правила частичного порядка (14.5.5.2) используются для определения является ли одна из специализаций более специализированной, чем другие. Если ни одна из специализаций является более специализированной, чем все другие соответствующие специализации, тогда использование шаблона класса неоднозначно, и программа плохо сформирована.

Частичный порядок соответствия шаблонов

https://stackoverflow.com/questions/17005985/what-is-the-partial-ordering-procedure-in-template-deduction

Пример

```
template <typename T> class X;
template <typename T> class X <T *>;
template <typename T> class X <const T *>;

// ...

X <const int *> x1;
X <int *> x2;
X <int> x3;
```

Зависимый тип

Определение (зависимый тип)

1. Параметр шаблона;
2. Член неизвестной специализации;
3. Вложенный класс или объединение — зависящий член от неизвестной специализации;
4. CV-квалификация зависимого типа;
5. Сложный тип, сконструированный с использованием зависимого типа;
6. Тип массива, сконструированный из зависимого типа или чей размер — константа, зависящая от значения;
7. Результат операции „**decltype** ()“, применённой к выражению, зависящему от типа.

Преобразования типов

`T`	`T(T)`	`T (T::*)()`
⟨cv-список⟩ `T`	`T` ⟨тип⟩ `::*`	`T (T::*)(T)`
`T *`	⟨тип⟩ `T::*`	⟨тип⟩ `[I]`
`T &`	`T T::*`	⟨имя шаблона кл.⟩ `<I>`
`T &&`	`T (`⟨тип⟩ `::*)()`	`TT <T>`
`T[`⟨целая константа⟩`]`	⟨тип⟩ `(T::*)()`	`TT <I>`
⟨имя шаблона кл.⟩ `<T>`	⟨тип⟩ `(`⟨тип⟩ `::*)(T)`	`TT <>`
⟨тип⟩ `(T)`	⟨тип⟩ `(T::*)(T)`	
`T()`	`T (`⟨тип⟩ `::*)(T)`	

Таблица 1: аргументы шаблона частичной специализации, зависящие от параметра-типа (`T`), не-типа `I`) и шаблона (`TT`)

Неизвестная специализация

Определение (член неизвестной специализации)

Квалификационное имя (`A <T>::k`) или выражение с операцией доступа к члену (`pA->k`), класс в которой зависим, и либо:

- не текущая конкретизация, либо:
- текущая, но имеет хоть 1 зависимый базовый класс, поиск имени после „`::`“ не находит ни 1 члена в текущем классе и его независимых базовых классах.

Неизвестная специализация

Пример

```
template <typename T> struct Base { };

template <typename T> struct Derived : Base <T>
{
  void f()
    { typename Derived <T>::unknown_type z; }
};

template <> struct Base <int>
{
  typedef int unknown_type;
};
```

Шаблонные псевдонимы типов

Более подробно об этой возможности в C++11:
https://en.cppreference.com/w/cpp/language/type_alias

Пример

```
template <typename T>
  struct AllocList
{
  typedef
    std::list <T, MyAlloc <T> >
    type;
};
```

Пример (окончание)

```
template <typename T>
  class Client
{
  // ...
private:
  typename AllocList <T>::type
    m_List;
};
```

Шаблонные псевдонимы типов

Более подробно об этой возможности в C++11:
https://en.cppreference.com/w/cpp/language/type_alias

Пример

```
template <typename T>
  using AllocList =
    std::list <T, MyAlloc <T>>;
```

Пример (окончание)

```
template <typename T>
  class Client
{
  // ...
private:
  AllocList <T> m_List;
};
```

Чистый функциональный язык

Признаки чистого функционального языка

- (Мета)данные неизменяемы.
- (Мета)функции не имеют побочных эффектов.

Отличия классов характеристик от функций

- Возможность специализации для отдельных значений или групп значений.
- Множественность возвращаемых значений.

Чистый функциональный язык

Признаки чистого функционального языка

- (Мета)данные неизменяемы.
- (Мета)функции не имеют побочных эффектов.

Отличия классов характеристик от функций

- Возможность специализации для отдельных значений или групп значений.
- Множественность возвращаемых значений.

Реализация обмена значений

Пример

```
template <class FwdIter1, class FwdIter2>
  void iter_swap(FwdIter1 i1, FwdIter2 i2)
{
  typename iterator_traits <FwdIter1>::value_type tmp = *i1;
  // или auto в C++11
  *i1 = *i2;     // или move(*i2)
  *i2 = tmp;
}
```

Блоб

C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. By David Abrahams

Определение

Блоб: (*Blob, Binary Large Object*) — класс с большим количеством тесно связанных описаний внутри себя.

Пример

```
template <class X, class Blob>
  X apply_fg(X x, Blob blob)
{
  return blob.f(blob.g(x));
}
```

Блоб

C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. By David Abrahams

Определение

Блоб: (*Blob, Binary Large Object*) — класс с большим количеством тесно связанных описаний внутри себя.

Пример

```
template <class X, class Blob>
  X apply_fg(X x, Blob blob)
{
  return blob.f(blob.g(x));
}
```

Блоб

C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. By David Abrahams

Определение

Блоб: (*Blob, Binary Large Object*) — класс с большим количеством тесно связанных описаний внутри себя.

Пример

```
template <class X, class UnaryOp1, class UnaryOp2>
  X apply_fg(X x, UnaryOp1 f, UnaryOp2 g)
{
  return f(g(x));
}
```

Использование раздельных аргументов

Пример

```
#include <functional>
#include <cmath>

// ...

int main()
{
  float f = apply_fg(0.5f, std::negate <float>(), std::sinf);
  // ...
}
```

Использование библиотеки характеристик типов

Пример (Boost)

```
#include <boost/type_traits.hpp>
// или #include <boost/type_traits/add_const.hpp> и т. п.

using namespace boost;
```

Пример (C++11)

```
#include <type_traits>

using namespace std;
```

Первичные характеристики типов

is_void	is_member_object_pointer
is_integral	is_member_function_pointer
is_floating_point	is_enum
is_array	is_union
is_pointer	is_class
is_lvalue_reference	is_function
is_rvalue_reference	

Таблица 2: первичные характеристики типов

Группирующие характеристики типов

`is_reference`	`is_scalar`
`is_arithmetic`	`is_compound`
`is_fundamental`	`is_member_pointer`
`is_object`	

Таблица 3: группирующие характеристики типов

Свойства типов

is_const	has_trivial_copy_constructor
is_volatile	has_trivial_assign
is_trivial	has_trivial_destructor
is_standard_layout	has_nothrow_default_constructor
is_pod	has_nothrow_copy_constructor
is_empty	has_nothrow_assign
is_polymorphic	has_virtual_destructor
is_abstract	alignment_of
is_signed	rank
is_unsigned	extent
has_trivial_default_constructor	

Таблица 4: свойства типов

Отношения между типами

`is_same`	`is_convertible`
`is_base_of`	

Таблица 5: отношения между типами

Преобразования типов

remove_const	add_rvalue_reference
remove_volatile	make_signed
remove_cv	make_unsigned
add_const	remove_extent
add_volatile	remove_all_extents
add_cv	remove_pointer
remove_reference	add_pointer
add_lvalue_reference	

Таблица 6: преобразования типов

Другие операции над типами

Операции

```
template <std::size_t Len, std::size_t Align = /* ... */>
  struct aligned_storage;
template <std::size_t Len, class ... Types> struct aligned_union;

template <class T> struct decay;

template <bool, class T = void> struct enable_if;
template <bool, class T, class F> struct conditional;

template <class ... Types> struct common_type;
template <class T> struct underlying_type;
template <class Fn, class ... ArgTypes>
  struct result_of <Fn(ArgTypes ...)>;
```

Стандартные объявления **using** (C++14)

Пример

```
template <typename T>
  void f1(T t)
{
  typename std::remove_reference <T>::type temp = t;
  // ...
}

template <typename T>
  void f2(T t)
{
  std::remove_reference_t <T> temp = t;
  // ...
}
```

Реализация обмена значений

Пример

```
template <class FwdIter1, class FwdIter2>
  void iter_swap(FwdIter1 i1, FwdIter2 i2)
{
  typename iterator_traits <FwdIter1>::value_type tmp = *i1;
  // или auto в C++11
  *i1 = *i2;     // или move(*i2)
  *i2 = tmp;
}
```

Реализация обмена значений через std::swap()

Пример

```
std::vector <std::list <std::string> > v1, v2;
// ...
iter_swap(v1.begin(), v2.begin());
// ...
```

Пример

```
template <class FwdIter1, class FwdIter2>
  void iter_swap(FwdIter1 i1, FwdIter2 i2)
{
  std::swap(*i1, *i2);
}
```

Реализация обмена значений через std::swap()

Пример

```
std::vector <std::list <std::string> > v1, v2;
// ...
iter_swap(v1.begin(), v2.begin());
// ...
```

Пример

```
template <class FwdIter1, class FwdIter2>
  void iter_swap(FwdIter1 i1, FwdIter2 i2)
{
  std::swap(*i1, *i2);
}
```

Реализация обмена значений с перегрузкой

Пример

```
template <class FwdIter1, class FwdIter2>
  void iter_swap(FwdIter1 i1, FwdIter2 i2)
{
  typename iterator_traits <FwdIter1>::value_type tmp = *i1;
  *i1 = *i2;
  *i2 = tmp;
}

template <class FwdIter>
  void iter_swap(FwdIter i1, FwdIter i2)
{
  std::swap(*i1, *i2);
}
```

Реализация std::swap()

Пример

```
std::vector <std::string> v1;
std::list <std::string> l1;
iter_swap(l1.begin(), v1.begin());
```

Пример

```
template <class T1, class T2> void swap(T1 &rT1, T2 &rT2)
{
  T1 tmp = rT1;
  rT1 = rT2;
  rT2 = tmp;
}
```

Реализация std::swap()

Пример

```
std::vector <std::string> v1;
std::list <std::string> l1;
iter_swap(l1.begin(), v1.begin());
```

Пример

```
template <class T1, class T2> void swap(T1 &rT1, T2 &rT2)
{
  T1 tmp = rT1;
  rT1 = rT2;
  rT2 = tmp;
}
```

Реализация std::swap()

Пример

```
std::vector <bool> v1, v2;
iter_swap(v1.begin(), v2.begin());
```

Пример

```
template <class T1, class T2> void swap(T1 &rT1, T2 &rT2)
{
  T1 tmp = rT1;
  rT1 = rT2;
  rT2 = tmp;
}
```

Реализация std::vector <bool>::reference

Определение «прокси-ссылки»

```
template <typename TAlloc>
  class vector <bool, TAlloc>
{
  // ...
  class reference
  {
    // ...
  public:
    operator bool() const;                          // if (v[i] = b) ...
    reference &operator = (const bool);             // v[i] = true;
    reference &operator = (const reference &);      // v[i] = v[j];
    void flip();                                    // v[i].flip();
    // ...
```

Реализация std::vector <bool>::reference (окончание)

Определение итератора

```
  // ...
  class iterator
  {
    // ...
  public:
    reference operator * () const;                    // *i ...
    // ...
  };
  // ...
  iterator begin();
  // ...
};    // vector <bool>
```

Вспомогательный класс для iter_swap()

Пример

```
template <bool>
  struct iter_swap_impl;

template <>
  struct iter_swap_impl <true>
{
  template <class FwdIter1, class FwdIter2>
    static void do_it(FwdIter1 i1, FwdIter2 i2)
  {
    std::swap(*i1, *i2);
  }
};
```

Вспомогательный класс для iter_swap() (окончание)

Пример

```
template <>
  struct iter_swap_impl <false>
{
  template <class FwdIter1, class FwdIter2>
    static void do_it(FwdIter1 i1, FwdIter2 i2)
  {
    typename iterator_traits <FwdIter1>::value_type tmp = *i1;
    *i1 = *i2;
    *i2 = tmp;
  }
};
```

Оптимизированная реализация iter_swap()

Пример

```
template <class FwdIter1, class FwdIter2>
  void iter_swap(FwdIter1 i1, FwdIter2 i2)
{
  typedef typename iterator_traits <FwdIter1>::value_type V1;
  typedef typename iterator_traits <FwdIter1>::reference  R1;
  typedef typename iterator_traits <FwdIter2>::value_type V2;
  typedef typename iterator_traits <FwdIter2>::reference  R2;
  const bool cbUseSwap =
    is_same <V1, V2>::value &&
    is_reference <R1>::value && is_reference <R2>::value;
  //
  iter_swap_impl <cbUseSwap>::do_it(*i1, *i2);
}
```

Неявное преобразование к value_type

Пример

```
template <typename T>
  void print_chars(std::ostream &rStream)
{
  // constexpr integral_constant::operator value_type ()
  if (is_arithmetic <T> ())  // вместо is_arithmetic <T> ()::value
    rStream << "arithmetic ";
  //
  // ...
}
```

Использование библиотеки enable_if

Операции

[lazy_]enable_if[_c]
[lazy_]disable_if[_c]

Пример (Boost)

```
#include <boost/utility/enable_if.hpp>

using namespace boost;
```

Определение enable_if[_c]

Определение enable_if_c

```
template <bool B, class T = void>
  struct enable_if_c
{
  typedef T type;
};

template <class T>
  struct enable_if_c <false, T>
{
  //
};
```

Определение enable_if

```
template
<
  class Cond,
  class T = void
>
  struct enable_if : public
    enable_if_c <Cond::value, T>
{
  //
};
```

Перегрузка функций (SFINAE)

Пример

```
template <typename T>
  typename enable_if <is_arithmetic <T>, T>::type ret(T t)
{
  // ...
  return (t + 1);
}

template <typename T>
  typename disable_if <is_arithmetic <T>, T>::type ret(T t)
{
  // ...
  return t;
}
```

Использование перегрузки функций

Пример

```
struct X {};

int main()
{
  int n;
  //
  ret(1);          //   арифметический тип
  ret(false);      //   арифметический тип
  ret(1.0f);       //   арифметический тип
  ret(&n);         // неарифметический тип
  ret(X());        // неарифметический тип
}
```

Перегрузка функций по параметру (SFINAE)

Пример

```
template <typename T>
  void pass(T t, typename enable_if <is_arithmetic <T> >::type * = 0)
{
  // ...
}

template <typename T>
  void pass(T t, typename disable_if <is_arithmetic <T> >::type * = 0)
{
  // ...
}
```

Использование перегрузки функций по параметру

Пример

```
struct X {};

int main()
{
  int n;
  //
  pass(1);          //   арифметический тип
  pass(false);      //   арифметический тип
  pass(1.0f);       //   арифметический тип
  pass(&n);         // неарифметический тип
  pass(X());        // неарифметический тип
}
```

Частичная специализация класса

Пример

```
template <class T, class Enable = void>
  class Data
{ /* ... */ };

template <class T>
  class Data <T, typename enable_if <is_integral <T> >::type>
{ /* ... */ };

template <class T>
  class Data <T, typename enable_if <is_float <T> >::type>
{ /* ... */ };
```

Ленивая конкретизация части описания функции

Пример

```
template <class T, class U>
  class mult_traits;

template <class T, class U>
  typename enable_if
  <
    is_multipliable <T, U>,
    typename mult_traits <T, U>::type
  >::type
    operator * (const T &rcT, const U &rcU)
  {
    // ...
  }
```

Ленивая конкретизация части описания функции

Пример

```
template <class T, class U>
  class mult_traits;

template <class T, class U>
  typename lazy_enable_if
  <
    is_multipliable <T, U>,
    typename mult_traits <T, U>
  >::type
    operator * (const T &rcT, const U &rcU)
  {
    // ...
  }
```

Описание типа **decltype** (C++11)

Пример

```
template <typename T1, typename T2>
  decltype (T1() * T2()) mult(T1 t1, T2 t2)
{
  return (t1 * t2);
}

int main()
{
  cout << mult(2, 2.6) << endl;
}
```