

Лекция 2. Простые примеры метапрограммирования

Метапрограммирование в C++

Преобразование двоичных литералов

Пример

```
#include <iostream>

unsigned binary(unsigned long ul)
{
    return (ul == 0 ? 0 : ul % 10 + 2 * binary(ul / 10));
}

int main()
{
    std::cout << binary(110101) << std::endl;
}
```

Преобразование двоичных литералов (Пролог)

Пример

```
binary(0, 0).
```

```
binary(B, D) :-  
    B1 is B // 10,  
    binary(B1, D1),  
    D is D1 << 1 \\/ (B mod 10).
```

Пример (запрос)

```
| ?- binary(110101, D).
```

```
D = 53 ?
```

Преобразование двоичных литералов (шаблоны)

Пример

```
template <unsigned long N>
    struct binary
    {
        static unsigned const value =
            binary <N / 10>::value << 1 | N % 10;
    };

template <>
    struct binary <0>
    {
        static unsigned const value = 0;
    };
```

Преобразование двоичных литералов (использование)

Пример

```
#include <iostream>

// ...

int main()
{
    const unsigned cuValue = binary <110101>::value;
    std::cout << cuValue << std::endl;
}
```

Преобразование двоичных литералов (C++11)

Пример

```
constexpr unsigned binary(unsigned long ul)
{
    return (ul == 0 ? 0 : ul % 10 + 2 * binary(ul / 10));
}

extern "C" int f()
{
    const unsigned cuValue = binary(110101);
    return cuValue;
}
```

Преобразование двоичных литералов (C++11)

Пример

```
> g++ -std=c++11 -c -S -o - -O2 test-constexpr.cpp
```

Преобразование двоичных литералов (C++11)

Пример

```
.file    "test-constexpr.cpp"
.text
.p2align 4,,15
.globl  f
.def    f;          .scl      2;          .type    32;          .endef
.seh_proc      f

f:
.LFB1:

.seh_endprologue
movl    $53, %eax
ret
.seh_endproc
.ident  "GCC: (tdm64-1) 5.1.0"
```


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

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

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

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 отдельных аргументов

Пример

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

// ...

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

Метафункции

Определения

Полиморфизм: способность кода работать с различными типами через единый интерфейс.

Метаданные: значения, доступные программе на этапе компиляции (константы целых типов, указателей на функции и т. д.) и **типы**.

Характеристики: (*traits*) — способ установления связей между различными метаданными при помощи специализации шаблонных классов.

Метафункция: (частный случай характеристик) — код, обрабатывающий метаданные на этапе компиляции. Реализуется при помощи класса или шаблона, получающего входные данные через шаблонные параметры (типы или шаблоны) и возвращающего результат при помощи вложенного (**public**) описания типа `type`.

Пример метафункции

Пример

```
template <typename T>
    struct make_const_param
{
    typedef const T &type;
};

template <>
    struct make_const_param <char>
{
    typedef char type;
};
```

Пример (использование)

```
template <typename T>
    struct Data
{
    void f(
        typename
            make_const_param <T>::type);
};

int main()
{
    Data <int> di;
    // ...
```

Числовые метафункции

Определения

Тип-обёртка над значением: (*wrapper type*) — тип, хранящий информацию о заданном значении в константном статическом поле (**public**) с именем `value`.

Числовая метафункция: (*numerical metafunction*) — метафункция, возвращающая тип-обёртку над значением (возможно, саму себя). Для удобства использования значение кроме `type::value` доступно также как поле `value`.

Пример числовой метафункции

Пример

```
template <typename T, T Val>
    struct integral_constant
    {
        static const T                value = Val;
        typedef T                    value_type;
        typedef integral_constant <T, Val> type;
        constexpr operator value_type () { return value; }
    };

typedef integral_constant <bool, true>  true_type;
typedef integral_constant <bool, false> false_type;
```

Пример числовой метафункции (окончание)

Пример

```
template <typename>
    struct is_lvalue_reference : public false_type
{
    //
};

template <typename T>
    struct is_lvalue_reference <T &> : public true_type
{
    //
};
```

Нуль-арная метафункция

Определения

Нуль-арная метафункция: (*nullary metafunction*) — метафункция, вызываемая без параметров (например, уже конкретизированная).

Пример

```
struct always_int
{
    typedef int type;
};
```

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

<code>is_void</code>	<code>is_member_object_pointer</code>
<code>is_integral</code>	<code>is_member_function_pointer</code>
<code>is_floating_point</code>	<code>is_enum</code>
<code>is_array</code>	<code>is_union</code>
<code>is_pointer</code>	<code>is_class</code>
<code>is_lvalue_reference</code>	<code>is_function</code>
<code>is_rvalue_reference</code>	

Таблица 1: первичные характеристики типов

Группирующие характеристики типов

is_reference	is_scalar
is_arithmetic	is_compound
is_fundamental	is_member_pointer
is_object	

Таблица 2: 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 3: свойства типов

Отношения между типами

is_same	is_convertible
is_base_of	

Таблица 4: отношения между типами

Преобразования типов

<code>remove_const</code>	<code>add_rvalue_reference</code>
<code>remove_volatile</code>	<code>make_signed</code>
<code>remove_cv</code>	<code>make_unsigned</code>
<code>add_const</code>	<code>remove_extent</code>
<code>add_volatile</code>	<code>remove_all_extents</code>
<code>add_cv</code>	<code>remove_pointer</code>
<code>remove_reference</code>	<code>add_pointer</code>
<code>add_lvalue_reference</code>	

Таблица 5: 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;
}
```