# Лекция 3. Метафункции библиотеки Boost MPL

## Метапрограммирование в C++

# Прямое определение абстрактной фабрики

## Пример (А. Александреску)

```
struct AbstractEnemyFactory
{
  virtual Soldier *CreateSoldier()
    = 0;
  virtual Monster *CreateMonster()
    = 0;
  virtual SuperMonster
    *CreateSuperMonster() = 0;
};
```

## Пример (окончание)

```
struct EasyLevelEnemyFactory :
  public AbstractEnemyFactory
{
  virtual Soldier *CreateSoldier()
  {
    return new SillySoldier();
  }
  // ...
};
```

# Использование библиотеки Loki

## Пример

```
#include <loki/AbstractFactory.h>

typedef Loki::AbstractFactory
<
  LOKI_TYPELIST_3(Soldier, Monster, SuperMonster)
> AbstractEnemyFactory;

typedef Loki::ConcreteFactory
<
  AbstractEnemyFactory,
  Loki::OpNewFactoryUnit,
  LOKI_TYPELIST_3(SillySoldier, SillyMonster, SillySuperMonster)
> EasyLevelEnemyFactory;
```

# Использование

## Пример

```
void process(AbstractEnemyFactory *pFactory)
{
  Soldier *pSoldier = pFactory->Create <Soldier> ();
  Monster *pMonster = pFactory->Create <Monster> ();
  // ...
}
```

# Использование библиотеки

## Пример

```
#include <boost/mpl/vector.hpp>
// #include <boost/mpl/⟨контейнер⟩.hpp>
#include <boost/mpl/int.hpp>
// #include <boost/mpl/⟨метафункция⟩.hpp>
// ...
#include <boost/mpl/placeholders.hpp>

namespace mpl = boost::mpl;
// или using namespace boost;
using namespace mpl::placeholders;
```

# Основные определения

## Определение

Продвижение метафункции: (*metafunction forwarding*) — создание метафункции при помощи открытого наследования от другой метафункции (для обеспечения вложенного определения `type`).

## Пример

```
template <unsigned long N>
  struct binary
{
private:
  struct digit : public mpl::int_ <N % 10> { };
```

# Основные определения

## Определение

Класс метафункции: (*metafunction class*) — класс с открытым вложенным определением метафункции с именем `apply`.

## Пример

```
struct add_pointer_f
{
  template <class T>
    struct apply
  {
    typedef typename boost::add_pointer <T>::type type;
  };
};
```

# Основные определения

## Определение

Класс метафункции: (*metafunction class*) — класс с открытым вложенным определением метафункции с именем `apply`.

## Пример

```
struct add_pointer_f
{
  template <class T>
    struct apply : boost::add_pointer <T> { };
};
```

# Метафункция высшего порядка

## Определение

Метафункция высшего порядка: (*higher-order metafunction*) — метафункция, обрабатывающая или возвращающая другие метафункции.

## Пример

```
template <class F, class X>
  struct twice
{
private:
  typedef typename F::template apply <X>::type once;        //
  f (x)
public:
  typedef typename F::template apply <once>::type type;     //
  f (f (x))
};
```

# Метафункция высшего порядка

## Определение

Метафункция высшего порядка: (*higher-order metafunction*) — метафункция, обрабатывающая или возвращающая другие метафункции.

## Пример

```
template <class F, class X>
  struct twice :
    F::template apply
    <
      typename F::template apply <X>::type
    >
  { };
```

# Использование

## Пример

```
int main()
{
  twice <add_pointer_f, int>::type ppn = NULL;
  // ...
}
```

# Использование метафункции высшего порядка

## Пример

```
template <class TUnaryMetaClass, class TArg>
  struct apply1
{
  typedef typename TUnaryMetaClass::template apply <TArg>::type type;
};
```

# Использование метафункции высшего порядка

## Пример

```
template <class TUnaryMetaClass, class TArg>
  struct apply1 : TUnaryMetaClass::template apply <TArg>
  { };
```

# Использование метафункции высшего порядка

## Пример

```
template <class TUnaryMetaClass, class TArg>
  struct apply1 : TUnaryMetaClass::template apply <TArg>
  { };
```

## Пример (использование)

```
template <class F, class X>
  struct twice : apply1 <F, typename apply1 <F, X>::type >
  { };
```

# Основные определения

## Определение

Заполнитель: (*placeholder*) — класс метафункции, возвращающей значение одного из своих аргументов.

## Определения (<boost/mpl/placeholders.hpp>)

```
template <int N> struct arg;
template <> struct arg <1>
{
  template <class A1, class A2 = void_, ..., class Am = void_>
    struct apply
  { typedef A1 type; };
};
typedef arg <1> _1;
```

# Выражение с заполнителями

## Определение

Выражение с заполнителями: (*placeholder expression*) — тип в одной из следующих форм:

- заполнитель;
- специализация шаблона с (хотя бы) одним из аргументов, являющимся выражением с заполнителями.

## Пример

```
if_ < less <_1, int_ <7> >, plus <_1, _2>, _1 >
```

# Выражение с заполнителями

## Определение

Выражение с заполнителями: (*placeholder expression*) — тип в одной из следующих форм:

- заполнитель;
- специализация шаблона с (хотя бы) одним из аргументов, являющимся выражением с заполнителями.

## Пример

```
if_ < less <_1, int_ <7> >, plus <_1, _2>, _1 >
```

# Использование выражений с заполнителями

## Пример

```
int main()
{
  twice    // !
  <
    boost::add_pointer <_1>, int
  >::type
    ppn = NULL;
  // ...
}
```

## Пример (конкретизация)

```
template <class T>
  struct add_pointer
{
  typedef T *type;
};

// add_pointer <_1>::type ~ "_1 *"
// add_pointer <_1>::apply ∄
```

# λ-выражения

## Определения

λ-выражение: (*lambda expression*) — вызываемые метаданные. Может быть в одной из следующих форм:

- класс метафункции;
- выражение с заполнителями.

Метафункция `lambda`: преобразует выражение с заполнителями в «соответствующий» класс метафункции, любой другой аргумент оставляет без изменения. Таким образом, преобразует любое λ-выражение в класс метафункции.

# λ-выражения

## Определения

λ-выражение: (*lambda expression*) — вызываемые метаданные. Может быть в одной из следующих форм:

- класс метафункции;
- выражение с заполнителями.

Метафункция `lambda`: преобразует выражение с заполнителями в «соответствующий» класс метафункции, любой другой аргумент оставляет без изменения. Таким образом, преобразует любое λ-выражение в класс метафункции.

# Использование λ-выражений

## Пример

```
#include <boost/type_traits.hpp>
#include <boost/mpl/lambda.hpp>
#include <boost/mpl/placeholders.hpp>

namespace mpl = boost::mpl;
using namespace mpl::placeholders;

int main()
{
  twice <mpl::lambda <boost::add_pointer <_1> >::type, int>::type
    ppn = NULL;
  // ...
}
```

# Определение «универсальных» метафункций

## Пример

```
template <class F, class X>
  struct twice :
    apply1
    <
      typename mpl::lambda <F>::type,
      typename apply1 <typename mpl::lambda <F>::type, X>::type
    >
  { };

int main()
{
  twice <add_pointer_f, int>::type ppn1 = nullptr;
  twice <boost::add_pointer <_1>, int>::type ppn2 = nullptr;
```

# Метафункция apply

## Определение

Метафункция `apply`: вызывает заданное $\lambda$-выражение с заданными аргументами.

## Пример

```
#include <boost/mpl/apply.hpp>
// ...

static const int g_cnValue =
  mpl::apply
  <
    mpl::plus <_1, _2>, mpl::int_ <6>, mpl::int_ <7>
  >::type::value;     // == 13
```

# Использовании метафункции apply

## Пример

```
template <class F, class X>
  struct twice :
    mpl::apply
    <
      typename mpl::lambda <F>::type,
      typename mpl::apply <typename mpl::lambda <F>::type, X>::type
    >
  { };
```

# Использовании метафункции apply

## Пример

```
template <class F, class X>
  struct twice :
    mpl::apply
    <
      F,
      typename mpl::apply <F, X>::type
    >
  { };
```

# Возможности $\lambda$-выражений

## Пример (частичная подстановка метафункции)

```
mpl::plus <_1, _1>
mpl::plus <_1, mpl::int_ <42> >
```

## Пример (композиция метафункций)

```
mpl::multiplies <mpl::plus <_1, _2>, mpl::minus <_1, _2> >
```

# Возможности $\lambda$-выражений

## Пример (частичная подстановка метафункции)

```
mpl::plus <_1, _1>
mpl::plus <_1, mpl::int_ <42> >
```

## Пример (композиция метафункций)

```
mpl::multiplies <mpl::plus <_1, _2>, mpl::minus <_1, _2> >
```

# Безымянный заполнитель

## Определение

**Безымянный заполнитель:** (*unnamed placeholder*) — заполнитель _. При преобразовании $\lambda$-выражения метафункцией `mpl::lambda` в класс метафункции каждое $n$-е вхождение безымянного заполнителя заменяется на заполнитель _n.

## Пример (семантика безымянных заполнителей)

| $\lambda$-выражение | Эквивалент |
|---|---|
| `mpl::plus <_, _>` | `mpl::plus <_1, _2>` |
| `is_same <_, add_pointer <_> >` | `is_same <_1, add_pointer <_1> >` |
| `multiplies < plus <_, _>, minus <_, _> >` | `multiplies < plus <_1, _2>, minus <_1, _2> >` |

# Правило обработки $\lambda$-выражений

## Правило

Если после замены заполнителей фактическими аргументами результирующая специализация X не имеет вложенного описания `type`, результатом $\lambda$-выражения является сам тип X.

## Пример

```
typedef mpl::apply <std::vector <_>, T>::type vector_of_t;
```

# Правило обработки $\lambda$-выражений

## Правило

Если после замены заполнителей фактическими аргументами результирующая специализация X не имеет вложенного описания type, результатом $\lambda$-выражения является сам тип X.

## Пример

```
typedef mpl::apply <std::vector <_>, T>::type vector_of_t;
```

# Правило обработки $\lambda$-выражений

## Правило

Если после замены заполнителей фактическими аргументами результирующая специализация X не имеет вложенного описания type, результатом $\lambda$-выражения является сам тип X.

## Пример

```
template <class U>
  struct make_vector
{
  typedef std::vector <U> type;
};

typedef mpl::apply <make_vector <_>, T>::type vector_of_t;
```

# Ленивое вычисление

## Определение

Ленивое вычисление: (*lazy evaluation*) — стратегия откладывания вычислений до момента, когда результат становится необходимым. Метафункция вычисляется в момент обращения к её результату (`type`).

## Пример

```
typedef mpl::vector <int, char *, double &> seq;
typedef mpl::transform <seq, boost::add_pointer <_> > calc_ptr_seq;
```

# Обёртки над интегральными типами

| | |
|---|---|
| `bool_ <C>` | `size_t <C>` |
| `int_ <C>` | `integral_c <T, C>` |
| `long_ <C>` | |

Таблица 1: обёртки над интегральными константами

## Определения обёрток

```
typedef bool_ <true>  true_;
typedef bool_ <false> false_;
```

# Первичные характеристики типов

## Определения обёрток

```
template <int N>
  struct int_
{
  static const int value = N;
  typedef int_ <N> type;
  typedef int value_type;
  typedef mpl::int_ <N + 1> next;
  typedef mpl::int_ <N - 1> prior;
  operator int () const { return N; }
};
```

# Проверка условия

## Пример

```
#include <boost/mpl/if.hpp>
#include <boost/type_traits/is_scalar.hpp>



template <class T> struct param_type :
  mpl::if_
  <
    typename boost::is_scalar <T>::type,
    T,
    const T &
  >
{ };
```

# Проверка условия

## Пример

```
#include <boost/mpl/if.hpp>
#include <boost/type_traits/is_scalar.hpp>
#include <boost/type_traits/add_reference.hpp>


template <class T> struct param_type :
  mpl::if_
  <
    typename boost::is_scalar <T>::type,
    T,
    typename boost::add_reference <const T>::type
  >
{ };
```

# Проверка условия

## Пример

```
#include <boost/mpl/if.hpp>
#include <boost/type_traits/is_scalar.hpp>
#include <boost/mpl/identity.hpp>
#include <boost/type_traits/add_reference.hpp>

template <class T> struct param_type :
  mpl::if_
  <
    typename boost::is_scalar <T>::type,
    mpl::identity <T>,
    typename boost::add_reference <const T>
  >::type
{ };
```

# Определение mpl::eval_if

## Определение mpl::eval_if

```
template <class C, class TrueMetafunc, class FalseMetafunc>
  struct eval_if :
    mpl::if_ <C, TrueMetafunc, FalseMetafunc>::type
{ };
```

# Основные определения

## Пример

```
#include <boost/mpl/eval_if.hpp>
#include <boost/type_traits/is_scalar.hpp>
#include <boost/mpl/identity.hpp>
#include <boost/type_traits/add_reference.hpp>

template <class T> struct param_type :
  mpl::eval_if
  <
    boost::is_scalar <T>,     // без typename...::type
    mpl::identity <T>,
    boost::add_reference <const T>
  >     // без ::type
{ };
```

# Основные определения

## Пример

```
// ...

template <class T> struct param_type :
  mpl::eval_if
  <
    mpl::or_
    <
      boost::is_scalar <T>,
      boost::is_stateless <T>,
      boost::is_reference <T>
    >,
    mpl::identity <T>, add_reference <const T>
  > { };
```

# Логические операции

| Специализация метафункции | ::value и ::type::value |
|---|---|
| `not_ <X>` | `!X::value` |
| `and_ <T1, T2, ..., Tn>` | `T1::value && ... && Tn ::value` |
| `or_ <T1, T2, ..., Tn>` | `T1::value \|\| ... \|\| Tn ::value` |

Таблица 2: логические операции

# Операции сравнения

| Специализация метафункции | ::value и ::type::value |
|---|---|
| equal_to <X, Y> | X::value == Y::value |
| not_equal_to <X, Y> | X::value != Y::value |
| greater <X, Y> | X::value > Y::value |
| greater_equal <X, Y> | X::value >= Y::value |
| less <X, Y> | X::value < Y::value |
| less_equal <X, Y> | X::value <= Y::value |

Таблица 3: операции сравнения

# Побитовые операции

| Специализация метафункции | ::value и ::type::value |
|---|---|
| bitand_ <X, Y> | X::value & Y::value |
| bitor_ <X, Y> | X::value \| Y::value |
| bitxor_ <X, Y> | X::value ^ Y::value |

Таблица 4: побитовые операции

# Арифметические операции

| Специализация метафункции | ::value и ::type::value |
|---|---|
| divides <T1, T2, ..., Tn> | T1::value / ... / Tn::value |
| minus <T1, T2, ..., Tn> | T1::value - ... - Tn::value |
| multiplies <T1, T2, ..., Tn> | T1::value * ... * Tn::value |
| plus <T1, T2, ..., Tn> | T1::value + ... + Tn::value |
| modulus <X, Y> | X::value % Y::value |
| shift_left <X, Y> | X::value << Y::value |
| shift_right <X, Y> | X::value >> Y::value |
| next <X> | X::next |
| prior <X> | X::prior |

Таблица 5: арифметические операции

# BOOST_STATIC_ASSERT

## Пример

```
#include <boost/static_assert.hpp>

template <unsigned long N>
  struct binary
{
private:
  static unsigned const digit = N % 10;
  BOOST_STATIC_ASSERT(digit == 0 || digit == 1);
public:
  static unsigned const value =
    (binary <N / 10>::value << 1) | digit;
};
```

# BOOST_STATIC_ASSERT (продолжение)

## Пример (gcc 5.1.0 без ключа -std=c++11)

```
error: invalid application of 'sizeof' to incomplete type
'boost::STATIC_ASSERTION_FAILURE<false>'
   BOOST_STATIC_ASSERT(digit == 0 || digit == 1);
   ^
```

# BOOST_STATIC_ASSERT (окончание)

## Определение BOOST_STATIC_ASSERT()

```
template <bool x> struct STATIC_ASSERTION_FAILURE;

template <> struct STATIC_ASSERTION_FAILURE <true>
  { enum { value = 1 }; };

#define BOOST_STATIC_ASSERT(B) \
   typedef ::boost::static_assert_test < \
      sizeof (::boost::STATIC_ASSERTION_FAILURE <(bool) (B)>) \
      > boost_static_assert_typedef_
```

# BOOST_MPL_ASSERT

## Пример

```
#include <boost/mpl/int.hpp>
#include <boost/mpl/or.hpp>
#include <boost/mpl/equal_to.hpp>
#include <boost/mpl/assert.hpp>

template <unsigned long N>
  struct binary
{
private:
  struct digit :
    public mpl::int_ <N % 10>
  { };
```

## Пример (окончание)

```
BOOST_MPL_ASSERT((
  mpl::or_
  <
    mpl::equal_to <
      digit, mpl::int_ <0> >,
    mpl::equal_to <
      digit, mpl::int_ <1> >
  >
));
// ...
```

# BOOST_MPL_ASSERT (окончание)

## Пример (gcc 5.1.0)

```
error: no matching function for call to
'assertion_failed(mpl_::failed************
boost::mpl::or_<boost::mpl::equal_to<binary<110102ul>::digit,
mpl_::int_<0> >, boost::mpl::equal_to<binary<110102ul>::digit,
mpl_::int_<1> >, mpl_::bool_<false>, mpl_::bool_<false>,
mpl_::bool_<false> >::************)'
          boost::mpl::assertion_failed<false>( \
                                               ^
```

# BOOST_MPL_ASSERT_NOT

## Пример

```
#include <boost/mpl/assert.hpp>
#include <boost/type_traits/is_integral.hpp>

template <typename T>
  class some
{
private:
  BOOST_MPL_ASSERT_NOT(boost::is_integral <T>);
  // ...
};
```

# BOOST_MPL_ASSERT_RELATION

## Пример

```
#include <boost/mpl/assert.hpp>

template <typename T, typename U>
  class data
{
private:
  BOOST_MPL_ASSERT_RELATION(sizeof (T), <=, sizeof (U));
  // ...
};

typedef data <int, char> dic_t;
dic_t dic;     // <-
```

# BOOST_MPL_ASSERT_RELATION (окончание)

## Пример (gcc 5.1.0)

```
error: no matching function for call to
'assertion_failed(mpl_::failed**********
mpl_::assert_relation<
  (mpl_::assert_::relations)6u, 4l, 1l>::**********)'
```

# BOOST_MPL_ASSERT_MSG

## Пример

```
#include <boost/mpl/assert.hpp>
#include <boost/type_traits.hpp>

template <typename T>
  struct data3
{
  BOOST_MPL_ASSERT_MSG(
    boost::is_integral <T>::value,
    NON_INTEGRAL_TYPES_ARE_NOT_ALLOWED, (types <T>));
  // ...
};

data3 <void> d3v;
```

# BOOST_MPL_ASSERT_MSG (окончание)

## Пример (gcc 5.1.0)

```
error: no matching function for call to
'assertion_failed(mpl_::failed**********
(data3<void>::NON_INTEGRAL_TYPES_ARE_NOT_ALLOWED::**********)
(mpl_::assert_::types<void, mpl_::na, mpl_::na, mpl_::na>))'
```

# static_assert (C++11)

## Пример

```
#include <type_traits>

template <typename T>
  struct data4
{
  static_assert(
    std::is_integral <T>::value,
    "Non-integral types are not allowed in template data4");
  // ...
};
```

# `static_assert` (окончание)

## Пример (gcc 5.1.0)

```
error: static assertion failed: Non-integral types are not
allowed in template data4
   static_assert(
   ^
```