Лекция 4. Контейнеры, алгоритмы и представления библиотеки Boost MPL

Метапрограммирование в C++

27 октября 2016 г.

Основные определения

Пример

```
#include <boost/mpl/vector.hpp>
#include <boost/mpl/find.hpp>

typedef mpl::vector <char, short, int, long, float, double> types;
typedef mpl::find <types, long>::type long_pos;
typedef
  mpl::if_
  <
    boost::is_same <long_pos, mpl::end <types> >,
    unsigned long,
    mpl::deref <long_pos>::type
  >::type my_type;
```

Прямые итераторы

Определение

Прямой итератор: (*Forward iterator*) — обладающий одним из взаимоисключающих свойств:

- увеличиваемость и разыменовываемость;
- указывает за конец последовательности.

Выражение	Результат	Предусловие
`mpl::next <I>::type`	Прямой итератор	`I` увеличиваемый
`mpl::deref <I>::type`	Любой тип	`I` разыменовываемый
`I::category`	Преобразуемый к `mpl::forward_iterator_tag`	

Таблица 1: свойства прямых итераторов

Двунаправленные итераторы

Определение

Двунаправленный итератор: (*Bidirectional iterator*) — прямой итератор, дополнительно обладающий одним из взаимоисключающих свойств:

- уменьшаемость;
- указывает на начало последовательности.

Выражение	Результат	Предусловие
`mpl::next <I>::type`	Двунаправленный итератор	`I` увеличиваемый
`mpl::prior <I>::type`	Двунаправленный итератор	`I` уменьшаемый
`I::category`	Преобразуемый к `bidirectional_iterator_tag`	

Таблица 2: дополнительные свойства двунаправленных итераторов

Итераторы произвольного доступа

Определение

Итератор произвольного доступа: (*Random access iterator*) — двунаправленный итератор, дополнительно обладающий возможностью перехода на любое количество позиций вперёд или назад и определение расстояния между итераторами за постоянное время.

Выражение	Результат
`mpl::next <I>::type`	Итератор произвольного доступа
`mpl::prior <I>::type`	Итератор произвольного доступа
`mpl::advance <I, N>::type`	Итератор произвольного доступа
`mpl::distance <I, J>::type`	Обёртка над интегральной константой
`I::category`	Преобразуем к `random_access_iterator_tag`

Таблица 3: дополнительные свойства итераторов произвольного доступа

Последовательности

Выражение	Результат
`mpl::begin <S>::type`	Итератор
`mpl::end <S>::type`	Итератор
`mpl::size <S>::type`	Обёртка над целой константой
`mpl::empty <S>::type`	Обёртка над логической константой

Таблица 4: свойства последовательностей

Категория последовательности	Выражение
Прямая	`mpl::front <S>::type`
Двунаправленная	`mpl::back <S>::type`
Произвольного доступа	`mpl::at <S, N>::type`

Таблица 5: свойства последовательностей

Расширяемые последовательности

Выражение
`mpl::insert <S, Pos, X>::type`
`mpl::insert_range <S, Pos, Range>::type`
`mpl::erase <S, Pos>::type`
`mpl::erase <S, First, Last>::type`
`mpl::clear <S>::type`

Таблица 6: свойства расширяемых последовательностей

Прямые/обратные расширяемые последовательности

Выражение
`mpl::push_front <S, X>::type`
`mpl::pop_front <S>::type`

Таблица 7: свойства прямых расширяемых последовательностей

Выражение
`mpl::push_back <S, X>::type`
`mpl::pop_back <S>::type`

Таблица 8: свойства обратных расширяемых последовательностей

Ассоциативные последовательности

Выражение
`mpl::has_key <S, K>::value`
`mpl::count <S, K>::type`
`mpl::at <S, K>::type`
`mpl::at <S, K, Def>::type`
`mpl::order <S, K>::type`
`mpl::key_type <S, T>::type`
`mpl::value_type <S, T>::type`

Таблица 9: свойства ассоциативных последовательностей

Расширяемые ассоциативные последовательности

Выражение
`mpl::insert <S, Pos, T>::type`
`mpl::insert <S, T>::type`
`mpl::erase <S, Pos>::type`
`mpl::erase_key <S, K>::type`
`mpl::clear <S>::type`

Таблица 10: свойства расширяемых ассоциативных последовательностей

mpl::list

Характеристика

Прямая последовательность, поддерживающая операции:

- `begin`
- `end`
- `front`
- `push_front`
- `pop_front`

mpl::vector

Характеристика

Последовательность произвольного доступа, дополнительно поддерживающая операции:

- back
- push_back
- pop_back

`mpl::deque`

Характеристика

Последовательность произвольного доступа, поддерживающая эффективные реализации для:

- `push_front`
- `pop_front`

mpl::range_c

Характеристика

«Ленивая» последовательность произвольного доступа, представляющая последовательность интегральных констант в заданном полуинтервале.
Поддерживаются операции:

- pop_front
- pop_back

Пример

```
#include <boost/mpl/range_c.hpp>
#include <boost/mpl/back.hpp>

typedef mpl::range_c <int, 0, 10> numbers;
const int g_cnLast = mpl::back <numbers>::type::value;     // 9
```

mpl::string

Характеристика

Двунаправленная расширяемая, прямая/обратная расширяемая последовательность интегральных констант, дополнительно поддерживающая операции:

- `c_str <S>::value`

Использование строк

Пример

```
typedef
  mpl::string
    <'Hell', 'o wo', 'rld'>
  hello;
typedef
  mpl::push_back
  <
    hello,
    mpl::char_ <'!'>
  >::type
  hello2;
```

Пример (окончание)

```
int main()
{
  std::cout
    << mpl::c_str <hello2>::value
    << std::endl;
  return 0;
}
```

Макросы, управляющие параметрами библиотеки

Пример

```
#define BOOST_MPL_LIMIT_STRING_SIZE 32
#define BOOST_MPL_LIMIT_VECTOR_SIZE 20
#define BOOST_MPL_LIMIT_METAFUNCTION_ARITY 5

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

Пример (вызов компилятора GCC)

```
gcc -fconstexpr-depth=512 -ftemplate-depth=1024 main.cpp
```

mpl::pair

Определения (<boost/mpl/pair.hpp>)

```
template <typename T1, typename T2>
  struct pair
{
  typedef pair type;
  typedef T1 first;
  typedef T2 second;
};
```

mpl::map

Характеристика

Ассоциативная последовательность, хранящая типы `mpl::pair`.

Пример

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

typedef mpl::map
<
  mpl::pair <bool, unsigned char>,
  mpl::pair <unsigned char, unsigned short>,
  mpl::pair <unsigned short, unsigned int>
  // ...
>::type to_larger;
```

mpl::set

Характеристика

Ассоциативная последовательность, хранящая произвольные типы.

Пример

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

typedef mpl::set <int, long, double, mpl::int_<5> > s;
BOOST_MPL_ASSERT((mpl::has_key <s, double>));
```

Преобразование последовательности в STL

Пример

```
#include <algorithm>

int main()
{
  std::vector <int> vi;
  std::list <float> vf;
  // ...
  std::copy(vi.begin(), vi.end(), std::back_inserter(vf));
  // ...
}
```

Класс вставки

Определение

Класс вставки: (*Inserter*) — структура, имеющая определения типов с именами:

- `state` — информация, передаваемая итерациям алгоритма;
- `operation` — бинарная операция для получения нового состояния (`state`) из элемента последовательности и текущего состояния (`state`).

Класс вставки (окончание)

Определения (<boost/mpl/inserter.hpp>)

```
template <class TState, class TOperation>
  struct inserter
{
  typedef TState state;
  typedef TOperation operation;
};
```

Пример

```
mpl::inserter <mpl::vector <>, mpl::push_back <_, _> >
```

Класс вставки (окончание)

Определения (<boost/mpl/inserter.hpp>)

```
template <class TState, class TOperation>
  struct inserter
{
  typedef TState state;
  typedef TOperation operation;
};
```

Пример

```
mpl::inserter <mpl::vector <>, mpl::push_back <_, _> >
```

Классы вставки

`front_inserter <S>`	`inserter <S, Op>`
`back_inserter <S>`	

Таблица 11: классы вставки в MPL

Пример

```
typedef
  mpl::copy
  <
    mpl::list <int, float, double>,
    mpl::back_inserter <mpl::vector <> >
  >::type result_vec;
```

Пример использования класса вставки

Пример

```
typedef
  mpl::push_back
  <
    mpl::push_back
    <
      mpl::push_back
      <
        mpl::vector <>, int
      >::type,
      float
    >::type,
    double
  >::type result_vec;
```

Пример использования класса вставки

Пример

```
typedef
  mpl::push_back
  <
    mpl::push_back
    <
      mpl::push_back
      <
        mpl::vector <>, int
      >::type,
      float
    >::type,
    double
  >::type result_vec;
```

Пример использования класса вставки

Пример

```
typedef
  mpl::push_back
  <
    mpl::push_back
    <



      mpl::vector <int>,
      float
    >::type,
    double
  >::type result_vec;
```

Пример использования класса вставки

Пример

```
typedef
  mpl::push_back
  <
    mpl::push_back
    <



      mpl::vector <int>,
      float
    >::type,
    double
  >::type result_vec;
```

Пример использования класса вставки

Пример

```
typedef
  mpl::push_back
  <
    mpl::vector <int, float>,
    double
  >::type result_vec;
```

Пример использования класса вставки

Пример

```
typedef
  mpl::push_back
  <
    mpl::vector <int, float>,
    double
  >::type result_vec;
```

Пример использования класса вставки

Пример

typedef

```
mpl::vector <int, float, double> result_vec;
```

Классы вставки по умолчанию

Пример

```
#include <boost/mpl/list.hpp>
#include <boost/mpl/equal.hpp>
#include <boost/mpl/copy.hpp>
#include <boost/mpl/assert.hpp>

typedef mpl::list <int, float, double> source_list;

typedef mpl::copy <source_list>::type result_list;

BOOST_MPL_ASSERT((mpl::equal <source_list, result_list>));
// BOOST_MPL_ASSERT((boost::is_same <source_list, result_list>));
```

Классы вставки по умолчанию (окончание)

Пример

```
typedef mpl::copy <source_list>::type
  result_list_1;

typedef
  mpl::reverse_copy
  <
    source_list,
    mpl::front_inserter <mpl::list <> >
  >::type
  result_list_2;
```

Обобщённые классы вставки

Пример

```
typedef
  mpl::vector
  <
    mpl::vector_c <int, 1, 2, 3>,
    mpl::vector_c <int, 4, 5, 6>,
    mpl::vector_c <int, 7, 8, 9>
  > seq;
```

Пример (окончание)

```
typedef
  mpl::transform
  <
    seq,
    mpl::front <_>,
    mpl::inserter
    <
      mpl::int_ <0>,
      mpl::plus <_, _>
    >
  >::type sum;
```

Алгоритмы std::accumulate()

Пример

```
#include <vector>
#include <numeric>

double func(double, double);

int main()
{
  std::vector <double> vd;
  // ...
  double dSumm = std::accumulate(vd.begin(), vd.end(), 0.);
  double dFold = std::accumulate(vd.rbegin(), vd.rend(), 3.14, func);
  // ...
}
```

Алгоритмы `mpl::fold` и `mpl::reverse_fold`

fold(*Seq*, *Prev*, *Op*) :–
если *Seq* = [ ] **то**
  вернуть *Prev*
иначе
  вернуть fold(хвост(*Seq*), Op(*Prev*, голова(*Seq*)), *Op*)

Рис. 1: алгоритм `mpl::fold`

reverse_fold(*Seq*, *Prev*, *Op*) :–
если *Seq* = [ ] **то**
  вернуть *Prev*
иначе
  вернуть Op(reverse_fold(хвост(*Seq*), *Prev*, *Op*), голова(*Seq*))

Рис. 2: алгоритм `mpl::reverse_fold`

Схемы алгоритмов `mpl::fold` и `mpl::reverse_fold`

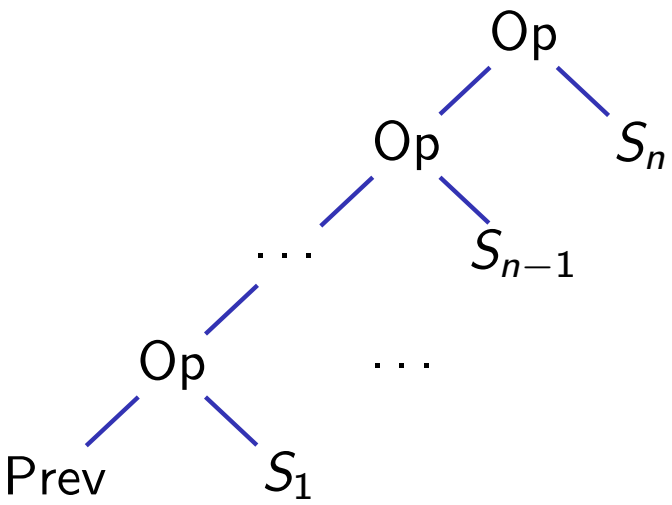

Рис. 3: схема алгоритма `fold`

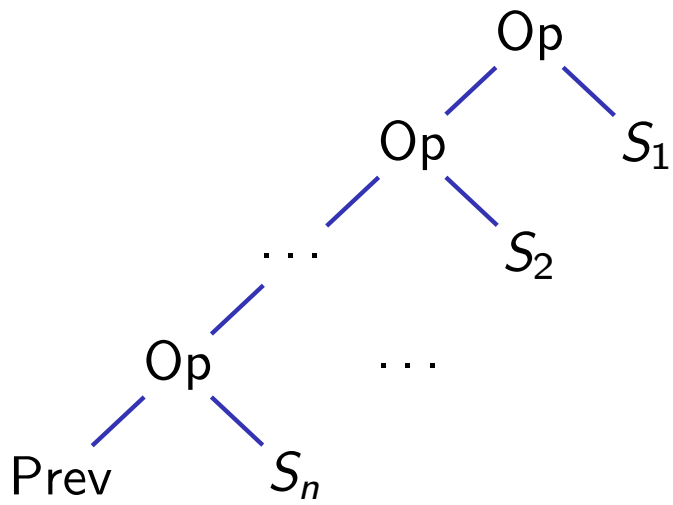

Рис. 4: схема алгоритма `reverse_fold`

Обобщение алгоритма `mpl::reverse_fold`

reverse_fold(*Seq*, *Prev*, *OpOut*, *OpIn* = *_1*) :–
если *Seq* = *[ ]* **то**
  вернуть *Prev*
иначе
  H ← голова(*Seq*)
  T ← хвост(*Seq*)
  вернуть OpOut(reverse_fold(*T*, OpIn(*Prev*, *H*), *OpOut*, *OpIn*), *H*)

Рис. 5: уточнённая версия алгоритма `mpl::reverse_fold`

Схема уточнённого алгоритма `mpl::reverse_fold`

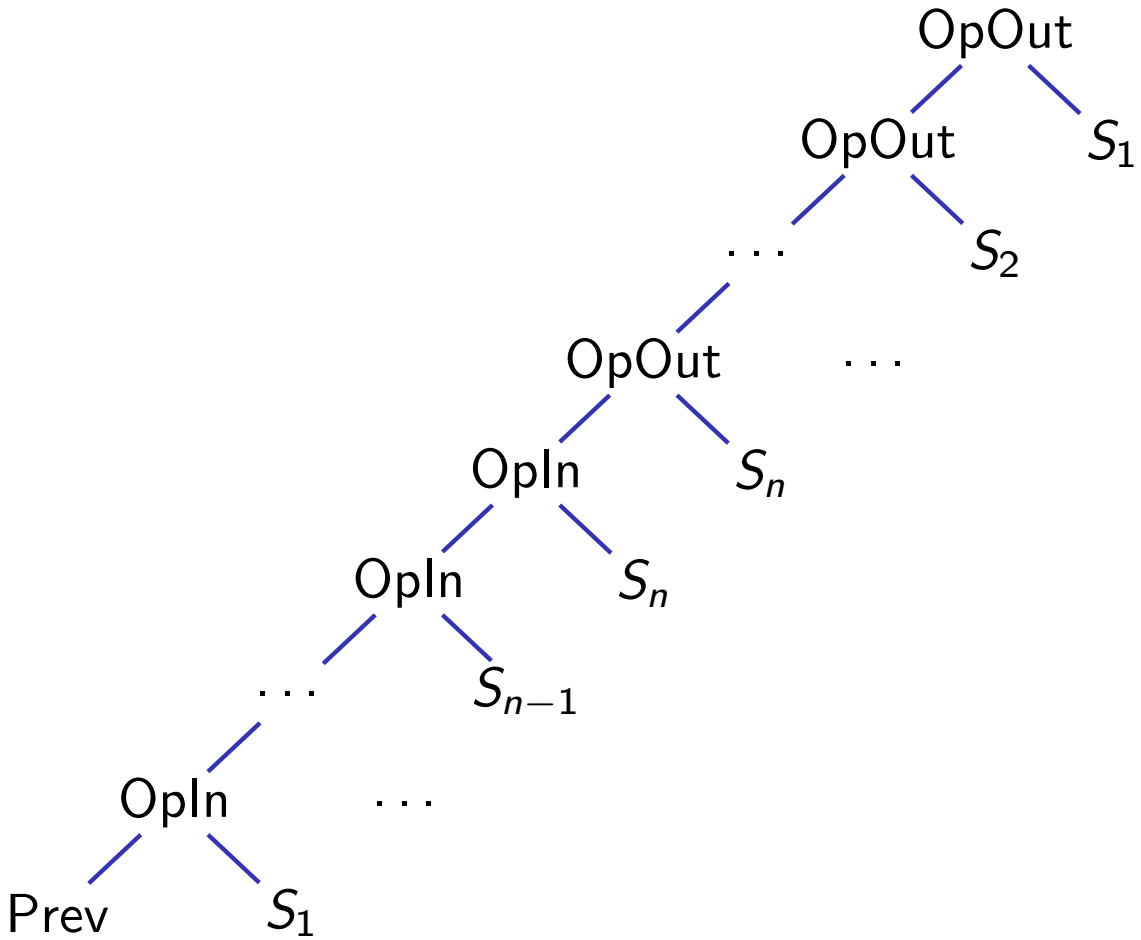

Рис. 6: схема уточнённого алгоритма `reverse_fold`

Определение mpl::reverse

Определение mpl::reverse

```
template <class TSeq>
  struct reverse :
    mpl::fold
    <
      TSeq,
      typename mpl::clear <TSeq>::type,
      mpl::push_front <_, _>
    >
{ };
```

Алгоритмы свёртки

`fold <Seq, Prev, OpFwd>`
`accumulate <Seq, Prev, OpFwd>`
`reverse_fold <Seq, Prev, OpBack, OpFwd = _1>`
`iter_fold <Seq, Prev, OpFwd>`
`reverse_iter_fold <Seq, Prev, OpBack, OpFwd = _1>`

Таблица 12: алгоритмы свёртки

Алгоритмы проверки

`find <Seq, T>`	`equal <Seq1, Seq2>`
`find_if <Seq, Pred>`	`lower_bound <Seq, T[, Comp]>`
`contains <Seq, T>`	`upper_bound <Seq, T[, Comp]>`
`count <Seq, T>`	`max_element <Seq[, Comp]>`
`count_if <Seq, Pred>`	`min_element <Seq[, Comp]>`

Таблица 13: алгоритмы проверки

Алгоритм mpl::find_if

Пример

```
typedef mpl::vector <char, short, int, long, float, double> types;

typedef
  mpl::find_if
  <
    types,
    mpl::greater <mpl::sizeof_ <_>, mpl::sizeof_ <wchar_t> >
  >::type pos_fit;

typedef mpl::deref <pos_fit>::type type_fit;
```

Правила алгоритмов построения последовательностей

Правила

- Для каждого алгоритма ⟨`алгоритм`⟩ определён обратный для него: `reverse_`⟨`алгоритм`⟩ (кроме `reverse`, обратного для `copy`).
- Последний аргумент алгоритма — необязательный класс вставки (для [`stable_`]`partition` — два аргумента).
- Если класс вставки не указан:
  - Если последовательность расширяема сзади, результат аналогичен указанию класса `back_inserter <clear <Seq>::type>`.
  - Если последовательность расширяема спереди, результат аналогичен указанию класса `front_inserter <clear <Seq>::type>` для обратного алгоритма.

Алгоритмы построения последовательностей

`copy <Seq>`	`remove <Seq, T>`
`copy_if <Seq, Pred>`	`remove_if <Seq, Pred>`
`transform <Seq1[, Seq2], OpUOrB>`	`unique <Seq[, Pred]>`
`replace <Seq, TOld, TNew>`	`[stable_]partition <Seq, Pred>`
`replace_if <Seq, Pred, TNew>`	`sort <Seq[, Pred]>`

Таблица 14: алгоритмы построения последовательностей

Алгоритм mpl::transform

Пример

```
typedef
  mpl::transform
  <
    types,
    mpl::if_
    <
      boost::is_signed <_>,
      boost::make_unsigned <_>,
      mpl::identity <_>
    >
  >::type types_unsigned;
```

Алгоритм mpl::transform (окончание)

Пример

```
template <typename Types>
  struct param_types :
    mpl::transform
    <
      Types,
      mpl::if_
      <
        mpl::bool_
        <
          boost::is_scalar <_>::value
          ||
          boost::is_array <_>::value
```

Пример (окончание)

```
        >,
        mpl::identity <_>,
        boost::add_reference
          <_>
      >
    >
{ };
```

Алгоритм mpl::transform (окончание)

Пример

```
template <typename Types>
  struct param_types :
    mpl::transform
    <
      Types,
      mpl::if_
      <
        mpl::or_
        <
          boost::is_scalar <_>,
          boost::is_array <_>
```

Пример (окончание)

```
          >,
          mpl::identity <_>,
          boost::add_reference
            <_>
        >
    >
{ };
```

Печать списка типов

Пример

```
#include <iostream>
#include <boost/mpl/for_each.hpp>

struct print_type
{
  template <class T>
    void operator () (T) const
  {
    std::cout
      << typeid (T).name() << " ";
  }
};
```

Пример (окончание)

```
template <class TSeq>
  void print_types()
{
  mpl::for_each <TSeq> (
    print_type());
  std::cout << std::endl;
}
```

Обход типов, не инициализируемых значением

Пример

```
typedef mpl::vector <int &, long &, char *&, void> Seq;
template <class T> struct wrap { };

struct print_type
{
  template <class T>
    void operator () (wrap <T>) const;    // T выводится
};

template <class TSeq> void print_types()
{
  mpl::for_each <mpl::transform <TSeq, wrap <_> >::type> (print_type());
}
```

Обход типов, не инициализируемых значением

Пример

```
typedef mpl::vector <int &, long &, char *&, void> Seq;
template <class T> struct wrap { };

struct print_type
{
  template <class T>
    void operator () (wrap <T>) const;    // T выводится
};

template <class TSeq> void print_types()
{
  mpl::for_each <TSeq, wrap <_> > (print_type());
}
```

Использование инспектора

Пример

```
struct visit_type
{
  template <class TVisitor>
    void operator () (
      TVisitor) const
  {
    TVisitor::visit();
  }
};
```

Пример (продолжение)

```
template <class T>
  struct print_visitor
{
  static void visit()
  {
    std::cout
      << typeid (T).name()
      << std::endl;
  }
};
```

Использование инспектора (окончание)

Пример (окончание)

```
template <class TSeq> void print_types()
{
  mpl::for_each <Seq, print_visitor <_> > (
    visit_type());
}
```

Представление

Определение

Представление последовательностей: (*Sequence view*) — адаптер интерфейса последовательности, элементы которого вычисляются лениво (по запросу).

Свойства представлений

- Представления не расширяемы.
- Итератор представления — адаптер итератора исходной последовательности (-стей), имеющий встроенное описание `::base`.

Представление mpl::filter_view

Пример

```
typedef mpl::vector <int, float, long, char [50], double> types;

typedef mpl::filter_view <types, boost::is_float <_> > types_float;

typedef
  mpl::max_element
  <
    types_float,
    mpl::less <mpl::sizeof_ <_1>, mpl::sizeof_ <_2> >
  >::type iter_max;

typedef mpl::deref <iter_max>::type type_max;
```

Представления последовательностей

`empty_sequence`	`filter_view <Seq, Pred>`
`single_view <T>`	`transform_view <Seq, F>`
`joint_view <Seq1, Seq2>`	`iterator_range <First, Last>`
`zip_view <SeqOfSeq>`	

Таблица 15: представления последовательностей

Поиск размера типа с **sizeof** (T) >= MinSize

Пример

```
template <class Seq, class MinSize>
  struct padded_size :
    mpl::deref
    <
      typename
        mpl::lower_bound
        <
          typename mpl::transform <Seq, mpl::sizeof_ <_> >::type,
          MinSize
        >::type
    >::type
{ };
```

Поиск размера типа с **sizeof** (T) >= MinSize

Пример

```
template <class Seq, class MinSize>
  struct padded_size_lazy :
    mpl::deref
    <
      typename
        mpl::lower_bound
        <
          mpl::transform_view <Seq, mpl::sizeof_ <_> >,
          MinSize
        >::type
    >::type
{ };
```

Суммирование по столбцам

Пример

```
typedef mpl::vector_c <int, 1, 2, 3, 4, 5> v1;
typedef mpl::vector_c <int, 5, 4, 3, 2, 1> v2;
typedef mpl::vector_c <int, 1, 1, 1, 1, 1> v3;
```

Суммирование по столбцам (окончание)

Пример

```
typedef
  mpl::transform_view
  <
    mpl::zip_view <mpl::vector <v1, v2, v3> >,
    mpl::plus
    <
      mpl::at <_, mpl::int_ <0> >,
      mpl::at <_, mpl::int_ <1> >,
      mpl::at <_, mpl::int_ <2> >
    >
  > sum;
```

Суммирование по столбцам (окончание)

Пример

```
typedef
  mpl::transform_view
  <
    mpl::zip_view <mpl::vector <v1, v2, v3> >,
    mpl::unpack_args <mpl::plus <_, _, _> >
  > sum;
```

Поиск типов

Пример

```
template <class Types>
  struct contains_int_or_cv_ref :
    mpl::contains
    <
      mpl::transform_view
      <
        Types,
        boost::remove_cv <boost::remove_reference <_> >
      >,
      int
    >
{ };
```

Факториалы

Пример

```
template <typename TN>
  struct factorial :
    mpl::if_
    <
      mpl::equal_to <TN, mpl::int_ <0> >,
      mpl::long_ <1>,
      mpl::multiplies
      <
        TN,
        factorial <mpl::minus <TN, mpl::int_ <1> > >
      >
    >::type
{ };
```

Факториалы (продолжение)

Пример

```
typedef
  mpl::range_c <int, 0, 10>
  Numbers;
```

Пример

```
typedef
  mpl::transform
  <
    Numbers,
    factorial <_>,
    mpl::back_inserter
    <
      mpl::vector <>
    >
  >::type
  Factorials;
```

Факториалы (окончание)

Пример

```
template <int N, int NMax>
  struct least_of_larger_factorial :
    mpl::lower_bound
    <
      mpl::transform_view    // не mpl::transform <>
      <
        mpl::range_c <int, 0, NMax>,
        factorial <_>
      >,
      mpl::int_ <N>
    >::type::base
{ };
```

Слияние и сортировка

Пример

```
template <typename TSeq1, typename TSeq2>
  struct merge_and_sort :
    mpl::sort
    <
      typename
        mpl::copy
        <
          mpl::joint_view <TSeq1, TSeq2>,
          mpl::back_inserter <mpl::vector <> >
        >::type
    >
{ };
```

Получение указываемых типов

Пример

```
template <typename Types>
  struct get_pointees :
    mpl::transform_view
    <
      mpl::filter_view
      <
        Types,
        boost::is_pointer <_>
      >,
      boost::remove_pointer <_>
    >
{ };
```