МГТУ
им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

Углубленное программирование на языке C++

Алексей Петров

Рекомендуемая литература

- Лишнер Р. STL. Карманный справочник. — Питер, 2005. — 188 с.
- Мюссер Д., Дердж Ж., Сейни А. C++ и STL. Справочное руководство. — Вильямс, 2010. — 432 с.
- Саммерфилд М. Qt. Профессиональное программирование. Разработка кроссплатформенных приложений на C++. — Символ-Плюс, 2011. — 560 с.
- Шлее М. Qt 4.8. Профессиональное программирование на C++. — БХВ-Петербург, 2012. — 894 с.
- Demming, R., Duffy, D.J. *Introduction to the Boost C++ Libraries; Volume I – Foundations* (Datasim Education BV, 2010).
- Demming, R., Duffy, D.J. *Introduction to the Boost C++ Libraries; Volume II – Advanced Libraries* (Datasim Education BV, 2012).
- Musser, D.R., Saini, A. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library* (Addison-Wesley, 1995).

Лекция №7. Функциональное программирование в C++11. Практическое введение в Boost. Требования к типам

1. C++11: параметризация алгоритмов STL лямбда-функциями и применение замыканий.
2. Состав и назначение Boost.
3. Примеры использования Boost: проверки времени компиляции, контейнеры, «умные» указатели.
4. Использование средств Boost для повышения производительности и безопасности кода.
5. Стандартные требования к типам.

Лямбда-функции и замыкания в языке C++11 (1 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Лямбда-функция — третий (наряду с указателями на функции и классами-функторами) вариант реализации функциональных объектов в языке C++11, обязанный своим названием Λ-исчислению — математической системе определения и применения функций, в которой аргументом одной функции (оператора) может быть другая функция (оператор).

Как правило, лямбда-функции являются **анонимными** и определяются **в точке их применения**.

Возможность присваивать такие функции переменным позволяет именовать их **лямбда-выражениями**.

Лямбда-функции и замыкания в языке C++11 (2 / 2)

Лямбда-функции (лямбда-выражения) могут использоваться **всюду**, где требуется **передача вызываемому объекту функции** соответствующего типа, в том числе — как фактические параметры обобщенных алгоритмов STL.

В лямбда-функции могут использоваться внешние по отношению к ней переменные, образующие **замыкание** такой функции.

Многие лямбда-функции весьма просты. Например, функция

```
[] (int x) { return x % m == 0; }
```

эквивалентна функции вида

```
bool foo(int x) { return x % m == 0; }
```

Основные правила оформления лямбда-функций

При преобразовании функции языка C++ в лямбда-функцию необходимо учитывать, что имя функции заменяется в лямбда-функции квадратными скобками [], а возвращаемый тип лямбда-функции:

- не определяется явно (слева);
- при анализе лямбда-функции с телом вида

  return expr;

  автоматически выводится компилятором как decltype(expr);
- при отсутствии в теле однооператорной лямбда-функции оператора return автоматически принимается равным void;
- в остальных случаях должен быть задан программистом при помощи «хвостового» способа записи:

```
[](int x) -> int { int y = x; return x – y; }
```

Ключевые преимущества лямбда-функций

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

- **Близость** к точке использования — анонимные лямбда-функции всегда определяются в месте их дальнейшего применения и являются единственным функциональным объектом, определяемым внутри вызова другой функции.
- **Краткость** — в отличие от классов-функторов немногословны, а при наличии имени могут использоваться повторно.
- **Эффективность** — как и классы-функторы, могут встраиваться компилятором в точку определения на уровне объектного кода.
- **Дополнительные возможности** — работа с внешними переменными, входящими в замыкание лямбда-функции.

Применение анонимных и именованных лямбда-функций


```
// для анонимных лямбда-функций:
std::vector<int> v1;
std::vector<int> v2; // ...
std::transform(v1.begin(), v1.end(),
       v2.begin(), [](int x) { return ++x; });

// для именованных лямбда-функций:
// тип lt10 зависит от реализации компилятора
auto lt10 = [](int x) { return x < 10; };
int cnt = std::count_if(
       v1.begin(), v1.end(), lt10);
bool b = lt10(300);  // b == false
```

Внешние переменные и замыкание лямбда-функций

Внешние по отношению к лямбда-функции **автоматические переменные**, определенные в одной с ней области видимости, могут захватываться лямбда-функцией и входить в ее **замыкание**. При этом в отношении доступа к переменным действуют следующие соглашения:

- [z] — доступ по значению к одной переменной (z);
- [&z] — доступ по ссылке к одной переменной (z);
- [=] — доступ по значению ко всем автоматическим переменным;
- [&] — доступ по ссылке ко всем автоматическим переменным;
- [&, z] , [=, &z], [z, &zz] — смешанный вариант доступа.

Внешние переменные и замыкание лямбда-функций: пример

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

```
int countN;
std::vector<double> vd; // ...
countN = std::count_if(vd.begin(), vd.end(),
       [](double x) { return x >= N; });
// эквивалентно
int countN = 0;
std::vector<double> vd; // ...
std::for_each(vd.begin(), vd.end(),
       [&countN](double x) {countN += x >= N;});
```

Библиотека Boost: общие сведения

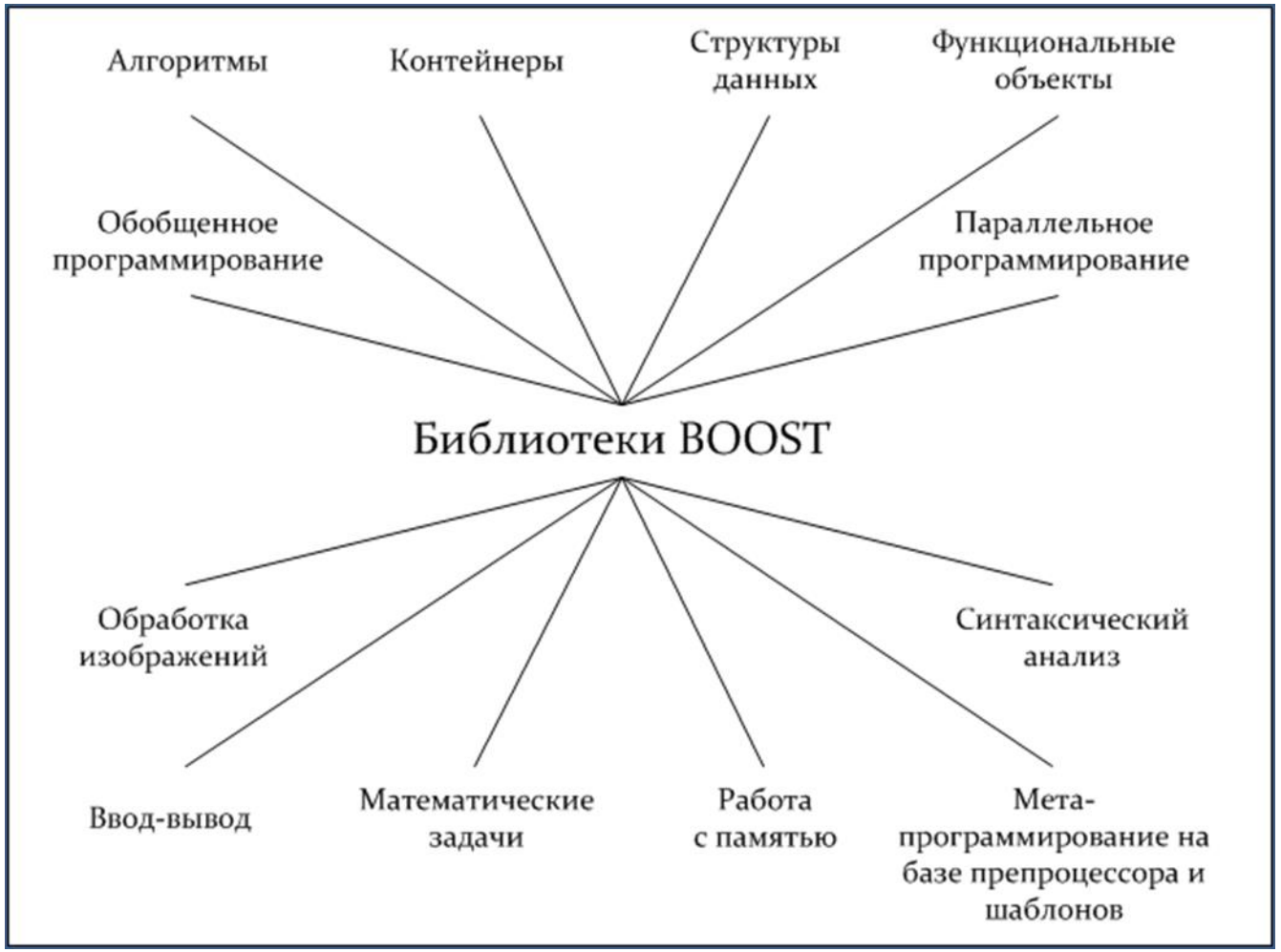
Boost — набор из более 80 автономных библиотек на языке C++, задуманный в 1998 г. Б. Давесом (Beman Dawes), Д. Абрахамсом (David Abrahams) и др.

Основными **целями разработки** Boost выступают:

- решение задач, выходящих за пределы возможностей стандартной библиотеки C++ в целом и STL — в частности;
- тестирование новых библиотек-кандидатов на включение в принимаемые и перспективные стандарты языка C++.

Преимущества и недостатки Boost связаны с активным использованием в Boost шаблонов и техник обобщенного программирования, что открывает перед программистами новые возможности, но требует немалой предварительной подготовки.

Состав и назначение Boost



Boost: примеры

Продемонстрируем работу Boost на следующих примерах:

- **внедрение в исходный код проверок времени компиляции** — позволяет не допустить компиляции логически или семантически неверного кода;
- **применение вариантных контейнеров и произвольных типов** — открывает возможность создания обобщенных и универсальных структур хранения данных;
- **применение циклических контейнеров** — дает возможность поддержки программной кэш-памяти и эффективных FIFO/LIFO-структур фиксированного размера;
- **использование «умных» указателей** — повышает качество кода в части работы с динамической памятью.

Пример 1. Проверки времени компиляции: общие сведения

Цель. Проверки времени компиляции (англ. static assertions) призваны предупредить случаи некорректного использования библиотек, ошибки при передаче параметров шаблонам и пр.

Библиотека.

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

Состав. Проверки времени компиляции представляют собой два макроопределения (x — целочисленная константа, msg — строка):

```
BOOST_STATIC_ASSERT(x)
BOOST_STATIC_ASSERT_MSG(x, msg)
```

являются **статическим аналогом** стандартного **макроопределения** assert и пригодны для применения **на уровне пространства имен, функции или класса**.

Пример 1. Проверки времени компиляции: реализация

Реализация. На уровне программной реализации в Boost макроопределения BOOST_STATIC_ASSERT* задействуют общий и полностью специализированный шаблон структуры вида:

```
namespace boost{
        template <bool>
        struct STATIC_ASSERTION_FAILURE;

        template <>
        struct STATIC_ASSERTION_FAILURE<true>{};
}
```

Пример 1. Проверки времени компиляции: использование

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

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

namespace my_conditions {
// проверка: длина int - не менее 32 бит
       BOOST_STATIC_ASSERT(
              std::numeric_limits<int>
              ::digits >= 32);
}
```

Пример 2. Вариантный контейнер: общие сведения

Цель. Предоставление безопасного обобщенного контейнера-объединения различимых типов со следующими возможностями:

- поддержка семантики значений, в том числе стандартных правил разрешения типов при перегрузке;
- безопасное посещение значений с проверками времени компиляции посредством boost::apply_visitor();
- явное извлечение значений с проверками времени выполнения посредством boost::get();
- поддержка любых типов данных (POD и не-POD), отвечающих минимальным требованиям (см. далее).

Библиотека.

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

Состав. Шаблон класса boost::variant с переменным числом параметров, сопутствующие классы и макроопределения.

Пример 2. Вариантный контейнер: требования к типам-параметрам

Обязательные характеристики типов-параметров шаблона boost::variant:

- наличие конструктора копирования;
- соблюдение безопасной по исключениям гарантии throw() для деструктора;
- полнота определения к точке инстанцирования шаблона boost::variant.

Желательные характеристики типов-параметров шаблона boost::variant:

- возможность присваивания (отсутствует для const-объектов и ссылок!);
- наличие конструктора по умолчанию;
- возможность сравнения по отношениям «равно» и «меньше»;
- поддержка работы с выходным потоком std::ostream.

Пример 2. Вариантный контейнер: определение и обход элементов


```
// создание и использование экземпляра
boost::variant< int, std::string > u("hello world");
std::cout << u << std::endl; // выдача: hello world

// для безопасного обхода элементов контейнера
// может использоваться объект класса-посетителя
// (см. далее):

boost::apply_visitor(
        times_two_visitor(), // объект-посетитель
        v                    // контейнер
);
```

Пример 2. Вариантный контейнер: класс-посетитель (1 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

```
class times_two_visitor
    : public boost::static_visitor<>
{
public:
    void operator()(int& i) const
    {
        i *= 2;
    }

    void operator()(std::string& str) const
    {
        str += str;
    }
};
```

Пример 2. Вариантный контейнер: класс-посетитель (2 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

```
// реализация класса-посетителя может быть обобщенной
class times_two_generic
    : public boost::static_visitor<>
{
public:
    template <typename T>
    void operator()( T& operand ) const
    {
        operand += operand;
    }
};
```

Пример 3. Произвольный тип: общие сведения

Цель. Предоставление безопасного обобщенного класса-хранилища единичных значений любых различимых типов, в отношении которых не предполагается выполнение произвольных преобразований. Основные возможности:

- копирование значений без ограничений по типам данных;
- безопасное проверяемое извлечение значения в соответствии с его типом.

Библиотека.

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

Состав. Шаблон класса boost::any, сопутствующие классы, в том числе производный от std::bad_cast класс boost::bad_any_cast, и другие программные элементы.

Пример 3. Произвольный тип: класс any


```
class any {
public:
// конструкторы, присваивания, деструкторы
        any();
        any(const any&);
        template<typename ValueType>
                any(const ValueType&);
        any& operator=(const any&);
        template<typename ValueType>
                any& operator=(const ValueType&);
        ~any();
// модификаторы
        any& swap(any&);
// запросы
        bool empty() const;
        const std::type_info& type() const;
};
```

Пример 3. Произвольный тип: работа со стандартными списками


```
// двусвязный список значений произвольных типов
// может формироваться и использоваться так:
typedef std::list<boost::any> many;

void append_string(many& values,
                   const std::string& value) {
    values.push_back(value);
}
void append_any(many& values,
                const boost::any& value) {
    values.push_back(value);
}
void append_nothing(many& values)
{
    values.push_back(boost::any());
}
```

Пример 3. Произвольный тип: проверка типов значений

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

```
bool is_int(const boost::any& operand) {
    return operand.type() == typeid(int);
}

bool is_char_ptr(const boost::any& operand)
{
        try {
                boost::any_cast<const char *>(operand);
                return true;
        }
        catch(const boost::bad_any_cast&) {
                return false;
        }
}
```

Пример 4. Циклический буфер: общие сведения

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Цель. Снабдить программиста STL-совместимым контейнером типа «кольцо» или «циклический буфер», который служит для приема поступающих данных, поддерживает перезапись элементов при заполнении, а также:

- реализует хранилище фиксированного размера;
- предоставляет итератор произвольного доступа;
- константное время вставки и удаления в начале и конце буфера;

Библиотека.

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

Состав. Шаблон класса boost::circular_buffer, адаптер boost::circular_buffer_space_optimized и другие программные элементы.

Пример 4. Циклический буфер: техника применения

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Использование. Циклический буфер и его адаптированный вариант на физическом уровне работают с непрерывным участком памяти, в силу чего не допускают неявных или непредсказуемых запросов на выделение памяти.

Возможные применения буфера включают, в том числе:

- хранение последних полученных (обработанных, использованных) значений;
- создание программной кэш-памяти;
- реализацию ограниченных буферов (англ. bounded buffer);
- реализацию FIFO/LIFO-контейнеров фиксированного размера.

Пример 4. Циклический буфер: порядок использования

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

```
boost::circular_buffer<int> cb(3);

cb.push_back(1);
cb.push_back(2);
cb.push_back(3);

// буфер полон, дальнейшая запись приводит
// к перезаписи элементов
cb.push_back(4); // значение 4 вытесняет 1
cb.push_back(5); // значение 5 вытесняет 2

// буфер содержит значения 3, 4 и 5

cb.pop_back();  // 5 выталкивается из посл. позиции
cb.pop_front(); // 3 выталкивается из нач. позиции
```

Пример 5. «Умные» указатели: общие сведения

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Цель. Реализовать набор сущностей, ответственных за хранение указателей на динамически размещаемые объекты и подобных стандартному указателю C++, но обладающих расширенными возможностями (поддержкой автоматического сбора мусора и т.д.):

- с теоретической точки зрения «умные» указатели являются **владеющими указателями** (**с подсчетом ссылок**) на управляемый объект, который будет гарантированно и своевременно удален из динамической памяти даже при возбуждении исключений (см. идиому RAII и шаблон ОО-проектирования «заместитель»).

Состав. Шаблоны классов-указателей, фабричные функции и другие программные элементы.

Использование. Тип T, являющийся параметром каждого из шести шаблонов классов-указателей, должен быть безопасен по исключениям при вызове T::~T() и T::operator delete().

Пример 5. «Умные» указатели: разновидности и библиотеки

Шаблон класса-указателя	Назначение указателя	Название библиотеки
`scoped_ptr`	Простой монопольный владелец единичного объекта. Без поддержки копирования	`<boost/scoped_ptr.hpp>`
`scoped_array`	Простой монопольный владелец массива. Без поддержки копирования	`<boost/scoped_array.hpp>`
`shared_ptr`	Владелец объекта, адресуемого множеством указателей	`<boost/shared_ptr.hpp>`
`shared_array`	Владелец массива, адресуемого множеством указателей	`<boost/shared_array.hpp>`
`weak_ptr`	Не имеющий прав владения наблюдатель за объектом во владении `shared_ptr`	`<boost/weak_ptr.hpp>`
`intrusive_ptr`	Немонопольный владелец объектов со встроенным подсчетом ссылок	`<boost/intrusive_ptr.hpp>`

Пример 5. «Умные» указатели: шаблон scoped_ptr (1 / 2)

Обоснование. Шаблон boost::scoped_ptr<T> является «простым решением простых задач», которое не поддерживает семантику совместного владения или передачи права владения, при этом:

- содержит указатель на объект, размещенный при помощи operator new;
- удаляет адресуемый объект при собственном удалении (RAII) или явном вызове метода reset().

Пример 5. «Умные» указатели: шаблон scoped_ptr (2 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Преимущества:

- высокая скорость выполнения операций, сравнимая с производительностью встроенных указателей;
- отсутствие накладных расходов оперативной памяти;
- больший уровень безопасности в сравнении с boost::shared_ptr<T> для указателей, которые не предполагают копирования.

Недостатки:

- невозможность использования в контейнерах STL — необходимо использование boost::shared_ptr<T>;
- неспособность хранить указатели на массивы — необходимо использование boost::scoped_array<T>.

Пример 5. «Умные» указатели: реализация шаблона scoped_ptr


```
template<class T> class scoped_ptr : noncopyable {
public:
        typedef T element_type;

        explicit scoped_ptr(T * p = 0);     // never throws
        ~scoped_ptr();                      // never throws

        void reset(T * p = 0);              // never throws

        T & operator*() const;              // never throws
        T * operator->() const;             // never throws
        T * get() const;                    // never throws

        operator unspecified-bool-type() const;
        // never throws
        void swap(scoped_ptr & b);          // never throws
};
template<class T> void swap(scoped_ptr<T> & a,
        scoped_ptr<T> & b);                 // never throws
```

1

2

Пример 5. «Умные» указатели: шаблон shared_ptr (1 / 2)

Обоснование. Шаблон boost::shared_ptr<T> поддерживает семантику совместного владения, при этом:

- содержит указатель на объект, размещенный при помощи operator new;
- удаляет адресуемый объект при удалении или явном сбросе последнего «умного» указателя на него (RAII);
- на уровне реализации использует подсчет ссылок.

Пример 5. «Умные» указатели: шаблон shared_ptr (2 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Преимущества:

- почти полное отсутствие требование к типу параметра шаблона (T);
- возможность использования в стандартных контейнерах C++;
- поддержка операции сравнения, необходимой для использования в ассоциативных контейнерах.

Недостатки:

- более низкий уровень безопасности в сравнении с boost::scoped_ptr<T> для указателей, которые не предполагают копирования;
- возможность появления «циклических указателей» — необходимо использование boost::weak_ptr<T>;
- неспособность хранить указатели на массивы — необходимо использование boost::shared_array<T> (до Boost 1.53).

Пример 5. «Умные» указатели: техника применения shared_ptr

Простейшей рекомендацией по использованию boost::shared_ptr<T>, практически устраняющей возможность утечек памяти, является использование именованных указателей для каждой операции new:

```
shared_ptr<T> p(new Y);   // типы T и Y могут не совпадать

shared_ptr<void> q(new int(42));
```

Такая техника сокращает количество явных операций delete и интенсивность использования пар try/catch.

Пример 5. «Умные указатели»: допустимые преобразования

В отсутствие каких-либо значимых ограничений на тип параметра shared_ptr<T> допускается:

- неявное преобразование shared_ptr<T> к shared_ptr<U>, если T* может быть неявно преобразовано в U*, к примеру, возможно:
- неявное преобразование shared_ptr<T> к shared_ptr<T const>;
- неявное преобразование shared_ptr<T> к shared_ptr<U>, если U есть достижимый из T базовый класс;
- неявное преобразование shared_ptr<T> к shared_ptr<void>.

Пример 5. «Умные» указатели: некорректное применение shared_ptr

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

```
void f(shared_ptr<int>, int);
int g();

void ok() {
        shared_ptr<int> p(new int(2));
        f(p, g());
}

void bad() {
        f(      // порядок вычисления не определен!!!
                shared_ptr<int>(      // может быть 3-м???
                        new int(2)    // может быть 1-м
                ),
                g()             // может быть 2-м: exception!!!
        );
}
```

Пример 5. «Умные» указатели: шаблон weak_ptr (1 / 2)

Обоснование. Шаблон boost::weak_ptr<T> содержит «слабую ссылку» на объект, находящийся во владении экземпляра boost::shared_ptr<T>, при этом:

- для доступа к адресуемому объекту экземпляр boost::weak_ptr<T> может быть преобразован в (подан на вход конструктора) boost::shared_ptr<T> или инициировать вызов своего метода lock();
- при обращении к уничтоженному адресуемому объекту первый вышеуказанный способ приводит к возбуждению исключения boost::bad_weak_ptr, а второй — к получению «пустого» экземпляра boost::shared_ptr<T>.

Пример 5. «Умные» указатели: шаблон weak_ptr (2 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Преимущество:

- устранение «циклических указателей».

Недостатки:

- ограниченность множества операций;
- потенциальная небезопасность обращения к адресуемому объекту (даже в однопоточном контексте).

```
shared_ptr<int> p(new int(5));
weak_ptr<int>   q(p);

// …

if(shared_ptr<int> r = q.lock()) { /* … */ }
```

Пример 5. «Умные» указатели: шаблон intrusive_ptr (1 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Обоснование. Шаблон boost::intrusive_ptr<T> реализует встроенный подсчет ссылок на объект, при этом:

- каждый новый экземпляр boost::intrusive_ptr<T> увеличивает число ссылок путем вызова незаданной (пользовательской) функции intrusive_ptr_add_ref(), принимающей указатель как аргумент;
- деструктор шаблона boost::intrusive_ptr<T> уменьшает число ссылок и вызывает незаданную (пользовательскую) функцию intrusive_ptr_release();
- адресуемый объект удаляется при достижении числом ссылок нулевого значения.

Возможные **причины использования** «умных» указателей такого рода, в числе прочего, связаны с предоставлением объектов с подсчетом ссылок каркасами разработки и современными ОС.

Пример 5. «Умные» указатели: шаблон intrusive_ptr (2 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Преимущества:

- отсутствие накладных расходов оперативной памяти;
- возможность создания экземпляра boost::intrusive_ptr<T> из произвольного встроенного указателя T*.

Недостаток:

- необходимость реализации функций intrusive_ptr_add_ref() и intrusive_ptr_release().

Boost: что еще? (1 / 2)

Boost Interval Container Library (ICL) — библиотека интервальных контейнеров с поддержкой множеств и отображений интервалов:

```
// работа с интервальным множеством
interval_set<int> mySet;
mySet.insert(42);
bool has_answer = contains(mySet, 42);
```

Boost.Tribool — поддержка тернарной логики «да, нет, возможно» :

```
tribool b(true);
b = false;
b = indeterminate;
```

Boost: что еще? (2 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Boost.Units — библиотека поддержки анализа размерностей (единиц измерения) операндов вычислительных операций. Задача анализа рассматривается как обобщенная задача метапрограммирования времени компиляции:

```
quantity<force>  F(2.0 * newton); // сила
quantity<length> dx(2.0 * meter); // расстояние
quantity<energy> E(work(F, dx));  // энергия
```

Математические библиотеки

Математическими библиотеками Boost, в частности, выступают:

- **Geometry** — решение геометрических задач (например, вычисление расстояния между точками в сферической системе координат);
- **Math Toolkit** — работа со статистическими распределениями, специальными математическими функциями (эллиптическими интегралами, гиперболическими функциями, полиномами Эрмита и пр.), бесконечными рядами и др.;
- **Quaternions** — поддержка алгебры кватернионов;
- **Ratio** — поддержка рациональных дробей (ср. std::ratio в C++11);
- **Meta State Machine** — работа с автоматными структурами;
- и др.

Требования стандартной библиотеки языка C++

Широкое применение типовых сочетаний требований к (алгоритмическим) характеристикам стандартных и пользовательских типов, в том числе со стороны элементов Boost, привело к появлению понятия **требований стандартной библиотеки**. Среди них:

- DefaultConstructible / Destructible — тип имеет конструктор по умолчанию / деструктор;
- CopyAssignable / CopyConstructible — тип поддерживает операцию присваивания с копированием / конструктор копирования;
- MoveAssignable / MoveConstructible — тип поддерживает операцию присваивания с переносом / конструктор переноса;
- Container — тип является структурой данных с доступом к элементам по итераторам.

Требование CopyConstructible

Тип, отвечающий требованию CopyConstructible, реализует одну или несколько следующих функций:

```
Type::Type( Type& other );
Type::Type( const Type& other );
Type::Type( volatile Type& other );
Type::Type( const volatile Type& other );
```

и гарантирует работоспособность следующих выражений, вычисление которых должно давать правильный, с языковой точки зрения, результат:

```
Type a = v;

Type(v);
```

Стандартные функции проверки соответствия требованиям

В стандартную библиотеку языка C++11 введены **шаблоны структур**, устанавливающие соответствие типов-параметров предъявляемым требованиям.

Например, для требования CopyConstructible:

```
template< class T > struct is_copy_constructible;

template< class T > struct is_trivially_copy_constructible;

template< class T > struct is_nothrow_copy_constructible;
```

Для интроспекции типов-параметров служит открытый статический константный атрибут шаблона структуры value.

Стандартные функции проверки соответствия требованиям: пример

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

```
struct foo {
    string _s; // атрибут с нетривиальным
               //string::string(const string&)
};

struct bar {
    int _n;
    bar(const bar&) = default; // тривиальный конструктор,
                               // безопасный по исключениям
};

// is_copy_constructible<foo>::value == true
// is_trivially_copy_constructible<foo>::value == false

// is_trivially_copy_constructible<bar>::value == true
// is_nothrow_copy_constructible<bar>::value == true
```

Требование Container

Объекты типа, отвечающего требованию Container, содержат другие объекты и отвечают за управление памятью, выделенной для хранения содержащихся в них объектов.

Пусть C — тип Container, T — тип элемента. Тогда:

- тип C реализует поддержку встроенных типов value_type, reference, const_reference, iterator, const_iterator, difference_type, size_type;
- тип C реализует операции создания пустого и непустого контейнера, присваивания, сравнения, возврата итераторов на начало (конец) и пр.;
- тип C отвечает требованиям DefaultConstructible, CopyConstructible, EqualityComparable, Swappable;
- тип T отвечает требованиям CopyInsertable, EqualityComparable, Destructible.

Спасибо за внимание

Алексей Петров