# Лабораторная работа №4

# Функциональные объекты (функторы)

### Понятие функтора

Функтор — это сокращение от "функциональный объект", представляющий собой конструкцию, позволяющую использовать объект класса как функцию. В C++ для определения функтора достаточно описать класс, в котором переопределена операция ().

В C++ функциональный объект создаётся с помощью класса, у которого перегружен operator():

class compare\_class

{ public:

bool **operator**()(int A, int B) { **return** (A < B); }

};

// объявление функции сортировки

template <class ComparisonFunctor>

void sort\_ints(int\* begin\_items, int num\_items,

ComparisonFunctor c);

int main()

{ int items[] = {4, 3, 1, 2};

compare\_class functor;

sort\_ints(items, **sizeof**(items)/**sizeof**(int), functor);

}

То, как из объекта образуется функция, легко показать на таком простом примере:

#include <iostream>

#include <vector>

using namespace std;

class summator : private vector<int>

{public:

summator(const vector<int>& ini)

{ for (auto x : ini) this->push\_back(x);}

int operator()(bool even)

{int sum = 0;

auto i = begin();

if (even) i++;

while (i < end()) { sum += \*i++;

if (i == end()) break;

i++;

}

return sum;

}

};

int main(void)

{ setlocale(LC\_ALL, "rus");

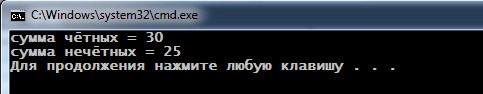
summator sums(vector<int>({ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }));

cout << "сумма чётных = " << sums(true) << endl

<< "сумма нечётных = " << sums(false) << endl;

}

Уже из такого простого примера видно следующее: операция () в классе может быть переопределена (точнее определена, поскольку она не имеет реализации по умолчанию) с произвольным числом, типом параметров и типом возвращаемого значения (или даже вовсе без возвращаемого значения). В итоге:



Выгода функтора состоит в том, что

а). его можно параметризовать при создании объекта (перед вызовом) используя конструктор объекта с параметрами и

б). может создаваться временный объект исключительно на время выполнения функционального вызова. Это иллюстрируется примером такого упрощённого целочисленного калькулятора:

#include <iostream>

using namespace std;

class calculate

{ char op;

public:

calculate( char op ) : op( op ) {}

int operator()( int op1, int op2 )

{ switch( op )

{ case '+': return op1 + op2;

case '-': return op1 - op2;

case '\*': return op1 \* op2;

case '/': return op1 / op2;

case '%': return op1 % op2;

case '^': { int ret = op1;

while( op2-- > 1 ) ret \*= op1;

return ret;

}

default: cout << "неразрешённая операция" << endl;

return 0;

}

}

};

int main( int argc, char \*\*argv, char \*\*envp )

{ setlocale(LC\_ALL, "rus");

char oper;

int op1, op2;

do

{cout << "выражение для вычисления (<op1><знак><op2>): "

<< flush;

cin >> op1 >> oper >> op2;

cout << op1 << ' ' << oper << ' ' << op2 << " = "

<< calculate( oper )( op1, op2 ) << endl;

} while( true );

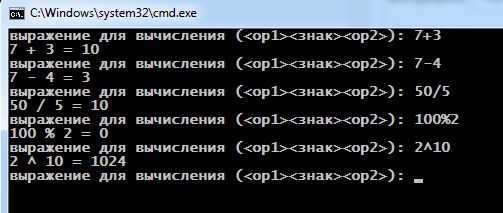
return 0;

}

Здесь в строке cout << calculate( oper )( op1, op2 ) действия выполняются последовательно:

* создаётся временный объект [класса](http://purecodecpp.com/archives/1703) calculate конструктором с параметром oper;
* для этого объекта выполняется метод () (функциональный вызов) с двумя параметрами;
* операция, которая будет выполнена в этом функциональном вызове, зависит от того параметра oper, с которым был сконструирован объект;
* функциональный вызов возвращает значение результата операции;
* созданный временный объект, сразу же после этого разрушается (если бы у него был описан [деструктор](http://purecodecpp.com/archives/1764), то он бы вызывался в этой точке).

И в итоге получаем:



Но особо широкое применение функторы приобрели в алгоритмах STL, когда они передаются в вызов в качестве параметра, вместо функции, определяющей действие или предикат алгоритма.

Функции и представляющие их объекты (функторы) занимают важное место в STL. Они используются ассоциативными контейнерами для упорядочения элементов, управляют работой алгоритмов типа find\_if, конструкции for\_each и transform без них теряют смысл. Умение создавать правильно работающие функторы абсолютно необходимо для эффективного использования STL.

### Проектируйте классы функторов для передачи по значению

Ни C, ни C++ не позволяют передавать функции в качестве параметров других функций. Вместо этого разрешается передавать *указатели* на функции. Например, объявление стандартной библиотечной функции qsort выглядит следующим образом:

void qsort(void \*base, size\_t nmemb, size\_t size,

**int (\*cmpfcn)(const void\*, const void\*)**);

Вместо функции qsort обычно рекомендуется использовать алгоритм sort, но нас сейчас интересует объявление параметра cmpfcn функции qsort. При внимательном анализе становится ясно, что аргумент cmpcfn, который является указателем на функцию, копируется (то есть передается по значению) из точки вызова в функцию qsort. Данный пример поясняет правило, соблюдаемое стандартными библиотеками C и C++, — указатели на функции должны передаваться по значению.

Объекты функций STL создавались по образцу указателей на функции, поэтому в STL также действует правило, согласно которому объекты функций передаются по значению (то есть копируются). Вероятно, это правило лучше всего демонстрирует приведенное в Стандарте объявление алгоритма for\_each, который получает и передает по значению объекты функций:

// Возврат по значению

template<class InputIterator, class Function> **Function**

// Передача по значению

for\_each(InputIterator first, InputIterator last, **Function f**);

Передача по значению не гарантирована полностью, поскольку вызывающая сторона может явно задать типы параметров в точке вызова. Например, в следующем фрагменте for\_each получает и возвращает функторы по ссылке:

class DoSomething:public unary\_function<int, void>

{ // Считаем, что базовый класс описан

void operator()(int x){…}

};

// Вспомогательное определение

typedef deque<int>::iterator DequeIntIter;

deque<int> di;

…

DoSomething d; // Создать объект функции

…

for\_each<**DequeIntIter**, // Вызвать for\_each с типами

**DoSomethng&**>(di.begin(), // параметров DequeIntIter

di.end(), // и DoSomething&; в результате

d); // происходит передача

// и возврат по ссылке.

Пользователи STL почти никогда не используют эту возможность, а в некоторых реализациях алгоритмов STL при передаче объектов функций по ссылке программы даже не компилируются. Будем считать, что объекты функций всегда передаются по значению, поскольку на практике это почти всегда так.

Поскольку объекты функций передаются и возвращаются по значению, необходимо позаботиться о том, чтобы объект функции правильно работал при передаче подобным способом (то есть копированием). Для этого необходимо соблюдение двух условий.

Во-первых, объекты функций должны быть небольшими, в противном случае копирование обойдется слишком дорого.

Во-вторых, объекты функций должны быть *мономорфными* (то есть не полиморфными), поэтому в них не могут использоваться виртуальные функции.

Второе требование связано с тем, что при передаче по значению объектов производных классов в параметрах базового класса происходит *отсечение:* в процессе копирования удаляются специализированные составляющие.

Бесспорно, эффективность является важным фактором, и предотвратить отсечение тоже необходимо, однако не все функторы малы и мономорфны. Одно из преимуществ объектов функций перед обычными функциями заключается в отсутствии ограничений на объем информации состояния.

Столь же нереалистичен и запрет на полиморфные функторы. Иерархическое наследование и динамическое связывание относятся к числу важнейших особенностей C++, и при проектировании классов функторов они могут принести такую же пользу, как и в других областях. Что такое классы функторов без наследования? C++ без «++». Итак, необходимы средства, которые бы позволяли легко передавать большие и/или полиморфные объекты функций с соблюдением установленного в STL правила о передаче функторов по значению.

Такие средства действительно существуют. Достаточно взять данные и/или полиморфные составляющие, которые требуется сохранить в классе функтора, перенести их в другой класс и сохранить в классе функтора указатель на этот новый класс. Рассмотрим пример создания класса полиморфного функтора с большим количеством данных:

template<typename T> // BPFC = "Big Polymorphic Functor class"

class BPFC: public unary\_function<T, void>

{ private:

Widget w; // Класс содержит большой объем данных,

int х; // поэтому передача по значению

… // была бы неэффективной

public:

virtual void operator()(const T& val) const;

… //Виртуальная функция создает проблему отсечения

};

Мы выделяем все данные и виртуальные функции в класс реализации и создаем компактный, мономорфный класс, содержащий указатель на класс реализации:

**template<typename T>** //Новый класс реализации

**class BPFCImpl {** //для измененного BPFC.

**private:**

**Widget w;** // Все данные, ранее находившиеся

**int х;** // в BPFC, теперь размещаются

… // в этом классе,

**virtual ~BPFCImpl();** // В полиморфных классах нужен

// виртуальный деструктор,

**virtual void operator()(const T& val) const;**

**friend class BPFC<T>;** // Разрешить BPFC доступ к данным

**};**

template<typename T>

class BPFC: // Компактная, мономорфная версия

public unary\_function<T, void>

{ private:

BPFCImpl<T>\* pImpl; // Все данные BPFC

public:

void operator()(const T& val) const;

{ //Функция не является виртуальной;

//вызов передается BPFCImpl

plImpl->operator()(val);

}

};

Реализация BFPC::operator() дает пример того, как должны строиться реализации всех виртуальных функций BPFC: они должны вызывать свои виртуальные «прототипы» из BPFCImpl. Полученный в результате класс функтора (BPFC) компактен и мономорфен, но при этом он предоставляет доступ к большому объему данных состояния и работает полиморфно.

С позиций STL прежде всего необходимо помнить о том, что классы функторов, использующие данную методику, должны поддерживать соответствующий механизм копирования. Если бы вы были автором приведенного выше класса BPFC, то вам пришлось бы позаботиться о том, чтобы копирующий конструктор выполнял осмысленные действия с объектом BPFCImpl, на который он ссылается.

В сущности, копирующий конструктор BPFC — единственное, о чем вам придется побеспокоиться в контексте данного примера, поскольку при передаче и получении функторов от функций STL всегда происходит копирование. Из этого вытекают два требования: компактность и мономорфизм.

### Реализуйте предикаты в виде «чистых» функций

Для начала разберемся с основными терминами.

• *Предикатом* называется функция, возвращающая тип bool (или другое значение, которое может быть автоматически преобразовано к bool). Предикаты широко используются в STL. В частности, функции сравнения в стандартных ассоциативных контейнерах представляют собой предикаты. Предикатные функции часто передаются в виде параметров таким алгоритмам, как find\_if, и различным алгоритмам сортировки.

• *«Чистой»* функцией называется функция, возвращаемое значение которой зависит только от параметров. Если f — «чистая» функция, а x и y — объекты, то возвращаемое значение f(x, y) может измениться только в случае изменения х или у.

В C++ все данные, используемые «чистыми» функциями, либо передаются в виде параметров, либо остаются постоянными на протяжении всего жизненного цикла функции (естественно, такие постоянные данные объявляются с ключевым словом const). Если бы данные, используемые «чистой» функцией, могли изменяться между вызовами, то вызов этой функции в разные моменты времени с одинаковыми параметрами мог бы давать разные результаты, что противоречит определению «чистой» функции.

Из сказанного должно быть понятно, что нужно сделать, чтобы предикаты были «чистыми» функциями.

• *Предикатным классом* называется класс функтора, у которого функция operator() является предикатом, то есть возвращает true или false. Как и следует ожидать, во всех случаях, когда STL ожидает получить предикат, может передаваться либо настоящий предикат, либо объект предикатного класса.

Объекты функций передаются по значению, поэтому при проектировании необходимо позаботиться о возможном копировании. Для объектов функций, являющихся предикатами, существует и другой аргумент в пользу специальной поддержки копирования. Алгоритмы могут создавать копии функторов и хранить их определенное время перед применением, причем некоторые реализации алгоритмов этим активно пользуются. Важнейшим следствием этого факта является то, что предикатные функции должны быть «чистыми».

Предположим, вы нарушили это ограничение. Ниже приведен плохо спроектированный класс предиката, который независимо от переданных аргументов возвращает true только один раз — при третьем вызове. Во всех остальных случаях возвращается false.

class BadPredicate: public unary\_function<Widget, bool>

{ public:

BadPredicate(): timesCalles(0) {} // Переменная timesCalled

// инициализируется нулем

bool operator()(const Widget&) {return ++timesCalled = 3;}

private:

size\_t timesCalled;

};

Предположим, класс BadPedicate используется для исключения третьего объекта Widget из контейнера vector<Widget>:

vector<Widget> vw; // Создать вектор и заполнить его

… // объектами Widget

vww.erase(**remove\_if(vw.begin(),** // Удалить третий объект Widget

**vw.end(),** remove\_if(**BadPredcate()**), vw.end());

Программа выглядит вполне разумно, однако во многих реализациях STL из вектора vw удаляется не только третий, но и шестой элемент!

Чтобы понять, почему это происходит, необходимо рассмотреть один из распространенных вариантов реализации remove\_if. Помните, что эта реализация *не является* обязательной.

template<typename FwdIterator, typename Predicate>

FwdIterator remove\_if(FwdIterator begin, FwdIterator end,

**Predicate p**)

{ begin = find\_if(begin, end, p);

if (begin==end) return begin;

else { FwdIterator next=begin;

return remove\_copy\_if(++next, end, begin, p);

}

}

Обратите внимание: предикат p сначала передается find\_if, а затем remove\_copy\_if. Конечно, в обоих случаях p передается по значению — то есть *копируется* (теоретически возможны исключения, но на практике дело обстоит именно так).

Первый вызов remove\_if (расположенный в клиентском коде, удаляющем третий элемент из vw) создает анонимный объект BadPredcate с внутренней переменной timesCalled, равной 0. Этот объект, известный в remove\_if под именем p, затем копируется в find\_if, поэтому find\_if тоже получает объект BadPredicate с переменной timesCalled, равной 0. Алгоритм find\_if «вызывает» этот объект, пока тот не вернет true; таким образом, объект вызывается три раза. Затем find\_if возвращает управление remove\_if. Remove\_if продолжает выполняться и в итоге вызывает remove\_copy\_if, передавая в качестве предиката очередную копию p. Но переменная timesCalled объекта p по-прежнему равна 0! Ведь алгоритм find\_if вызывал не p, а лишь *копию* p. В результате при третьем вызове из remove\_copy\_if предикат тоже вернет true. Теперь понятно, почему remove\_if удаляет два объекта Widget вместо одного.

Чтобы обойти эту лингвистическую ловушку, проще всего объявить функцию operator() с ключевым словом const в предикатном классе. В этом случае компилятор не позволит изменить переменные класса:

class BadPredicate: public unary\_function<Widget, bool>

{public:

bool operator()(const Widget&) **const**

{return ++timesCalled == 3; //Ошибка! Изменение локальных

} // данных в константной функции невозможно

};

Даже константные функции могут обращаться к mutablе-переменным, неконстантным локальным статическим объектам, неконстантным статическим объектам класса, неконстантным объектам в области видимости пространства имен и неконстантным глобальным объектам. Хорошо спроектированный предикатный класс должен обеспечить независимость функций operator() и от этих объектов. Объявление константных функций operator() в предикатных классах *необходимо* для правильного поведения, но *не достаточно.* Правильно написанная функция operator() является константной, но это еще не все. Она должна быть «чистой» функцией.

Ранее уже упоминалось о том, что всюду, где STL ожидает получить предикатную функцию, может передаваться либо реальная функция, либо объект предикатного класса. Этот принцип действует в обоих направлениях. В любом месте, где STL рассчитывает получить объект предикатного класса, подойдет и предикатная функция. Теперь вы знаете, что функции operator() в предикатных классах должны быть «чистыми» функциями, поэтому ограничение распространяется и на предикатные функции. Следующая функция также плоха в качестве предиката, как и объекты, созданные на основе класса BadPredcate:

bool anotherBadPredicate(const Widget&, const Widget&)

{ static int timesCalled = 0; // Нет! Нет! Нет! Нет! Нет! Нет!

return ++timesCalled == 3; // Предикаты должны быть "чистыми"

} // функциями, а "чистые" функции

// не имеют состояния

Как бы ни программировались предикаты, они всегда должны быть «чистыми» функциями.

### Классы функторов должны быть адаптируемыми

Предположим, у нас имеется список указателей Widget\* и функция, которая по указателю определяет, является ли объект Widget «интересным»:

list<Widget\*> WidgetPtrs;

bool isInteresting(const Widget \*pw);

Если потребуется найти в списке первый указатель на «интересный» объект Widget, это делается легко:

list<Widget\*>::iterator i = find\_if(widgetPts.begin(),

widgetPts.end(), isIntersting);

if (i != widgetPts.end())

{

… // Обработка первого "интересного" указателя на Widget

}

С другой стороны, если потребуется найти первый указатель на «неинтересный» объект Widget, следующее очевидное решение не компилируется:

list<Widget\*>::iterator i = find\_if(widgetPtrs.begin(),

widgetPtrs.end(),

**not1**(isInteresting)); // Ошибка! He компилируется

Перед not1 к функции isInteresting необходимо применить ptr\_fun:

list<Widget\*>::iterator i =

find\_if(widgetPtrs.begin(), widgetPtrs.end(),

not1(**ptr\_fun**(isInteresting))); // Нормально

if (i != widgetPtrs.end()) { // Обработка первого

… // "неинтересного" указателя

} //на Widget

При виде этого решения невольно возникают вопросы. *Почему* мы должны применять ptr\_fun к isInteresting перед not1? Что ptr\_fun для нас делает и почему начинает работать приведенная выше конструкция?

Ответ оказывается весьма неожиданным. Вся работа ptr\_fun сводится к предоставлению нескольких определений типов. Эти определения типов необходимы для not1, поэтому применение not1 к ptr\_fun работает, а непосредственное применение not1 к isInteresting не работает. Примитивный указатель на функцию isInteresting не поддерживает определения типов, необходимые для not1.

Впрочем, not1 — не единственный компонент STL, предъявляющий подобные требования. Все четыре стандартных адаптера (not1, not2, bind1st (устаревший) и bind2nd(устаревший)), а также все нестандартные STL-совместимые адаптеры из внешних источников (например, входящие в SGI и Boost), требуют существования некоторых определений типов. Объекты функций, предоставляющие необходимые определения типов, называются *адаптируемыми;* при отсутствии этих определений объект называется *неадаптируемым****.***Адаптируемые объекты функций могут использоваться в контекстах, в которых невозможно использование неадаптируемых объектов, поэтому следует по возможности делать свои объекты функций адаптируемыми. Адаптируемость не требует никаких затрат, но значительно упрощает использование классов функторов клиентами.

Наверное, вместо туманного выражения «некоторые определения типов» предпочтительней иметь точный список? Речь идет об определениях argument\_type, first\_argument\_type, second\_argument\_type и result\_type, но ситуация осложняется тем, что разные классы функторов должны предоставлять разные подмножества этих имен. Честно говоря, если вы не занимаетесь разработкой собственных адаптеров, вам вообще ничего не нужно знать об этих определениях. Как правило, определения наследуются от базового класса, а говоря точнее — от базовой структуры. Для классов функторов, у которых operator() вызывается с одним аргументом, в качестве предка выбирается структура std::unary\_function. Классы функторов, у которых operator() вызывается с двумя аргументами, наследуют от структуры std::binary\_function.

Впрочем, не совсем так. unary\_function и binary\_function являются шаблонами, поэтому прямое наследование от них невозможно. Вместо этого при наследовании используются структуры, созданные на основе этих шаблонов, а для этого необходимо указать аргументы типов. Для unary\_function задается тип параметра, получаемого функцией operator() вашего класса функтора, а также тип возвращаемого значения. Для binary\_function количество типов увеличивается до трех: типы первого и второго параметров operator() и тип возвращаемого значения.

Пара примеров:

template<typename T>

class MeetsThreshold:

**public std::unary\_function<Widget, bool>**

{private:

const T threshold;

public:

Meets Threshold(const T& threshold);

bool operator()(const WidgetS) const;

…

};

struct WidgetNameCompare:

**std::binary\_function<Widget, Widget, bool>**

{

bool operator()(const Widget& lhs, const Widget& rhs) const;

};

В обоих случаях типы, передаваемые unary\_function или binary\_function, совпадают с типами, получаемыми и возвращаемыми функцией operator() класса функтора, хотя на первый взгляд несколько странно, что тип возвращаемого значения operator() передается в последнем аргументе unary\_function или binary\_function.

Возможно, вы заметили, что MeetsTheshold является классом, а WidgetNameCompare является структурой. MeetsTheshold обладает внутренним состоянием (переменная threshold), и для инкапсуляции этих данных логично воспользоваться именно классом. WidgetNameCompare состояния не имеет, поэтому и закрытые данные не нужны. Авторы классов функторов, в которых вся информация является открытой, часто объявляют структуры вместо классов — вероятно, только для того, чтобы им не приходилось вводить «public» перед базовым классом и функцией operator(). Выбор между классом и структурой при объявлении таких функторов определяется исключительно стилем программирования. Следует учесть, что классы функторов без состояния в самой библиотеке STL (например, less<T>, plus<T> и т. д.) обычно записываются в виде структур.

Вернемся к определению WidgetNameCompare:

struct WidgetNameCompare:

std::binary\_function<**Widget, Widget**, bool>

{

bool operator()(const Widget& lhs, const Widget& rhs) const;

};

Если operator() получает параметры-указатели, ситуация меняется. Ниже приведена структура, аналогичная WidgetNameCompare, но работающая с указателями Widget\*:

struct PtrWidgetNameCompare:

std::binary\_function**<const Widget\*, const Widget\***, bool>

{

bool operator()(**const Widget\*** lhs, **const Widget\*** rhs) const;

};

В этом случае типы, передаваемые binary\_function, совпадают с типами, передаваемыми operator(). Общее правило для классов функторов, получающих или возвращающих указатели, заключается в том, что unary\_function или binary\_function передаются в точности те типы, которые получает или возвращает operator().

Помните, что базовые классы unary\_function и binary\_function выполняют только одну важную функцию — они предоставляют определения типов, необходимые для работы адаптеров, поэтому наследование от этих классов порождает адаптируемые объекты функций. Это позволяет использовать в программах следующие конструкции:

list<Widget> widgets;

…

list<Widget>::reverse\_iterator i1 = // Найти последний объект

find\_if(widgets.rbegin(), widgets.rend(), //Widget, not1(MeetsThreshold<int>(10))); // не соответствующий

// пороговому критерию 10(что бы это ни означало)

Widget w(*аргументы конструктора*); // Найти первый объект Widget,

list<Widget>::iterator i2 = // предшествующий w в порядке

find\_if(widgets.begin(), widgets.end(), // сортировки,

bind2nd(WidgetNameCompare(),w));//определенном WidgetNameCompare

Если бы классы функторов не определялись производными от unary\_function или binary\_function, ни один из этих примеров не компилировался бы, поскольку not1 и bind2nd работают только с адаптируемыми объектами функций.

Объекты функций STL построены по образцу функций C++, а функции C++ характеризуются единственным набором типов параметров и одним типом возвращаемого значения. В результате STL неявно подразумевает, что каждый класс функтора содержит единственную функцию operator(), типы параметров и возвращаемого значения которой должны передаваться unary\_function или binary\_function (с учетом правил передачи ссылок и указателей, о которых говорилось ранее). Из этого следует одно важное обстоятельство: не пытайтесь объединять функциональность WidgetnNameCompare и PtrWidgetCompare в одной структуре с двумя функциями operator(). В этом случае функтор будет адаптируемым по отношению лишь к одной из двух форм вызова (той, что использовалась при передаче параметров binary\_function), а пользы от такого решения будет немного — наполовину адаптируемый функтор ничуть не лучше неадаптируемого.

Иногда в классе функтора бывает разумно определить несколько форм вызова, тем самым отказавшись от адаптируемости, но это скорее исключение, а не правило. Адаптируемость важна, и о ней следует помнить при разработке классов функторов.

### Для чего нужны ptr\_fun, mem\_fun и mem\_fun\_ref

Загадочные функции ptr\_fun/mem\_fun/mem\_fun\_ref часто вызывают недоумение. В одних случаях их присутствие обязательно, в других они не нужны… но что же они все-таки делают? На первый взгляд кажется, что они бессмысленно загромождают имена функций. Их неудобно вводить и читать, они затрудняют понимание программы. Что это — очередные пережитки прошлого STL или синтаксическая шутка, придуманная членами Комитета по стандартизации?

Действительно, имена выглядят довольно странно, но функции ptr\_fun, mem\_fun и mem\_fun\_ref выполняют важные задачи. Если уж речь зашла о синтаксических странностях, надо сказать, что одна из важнейших задач этих функций связана с преодолением синтаксической непоследовательности C++.

В C++ существуют три варианта синтаксиса вызова функции f для объекта x:

f(x); // **Синтаксис 1**: f не является функцией класса

//(вызов внешней функции)

x.f(); // **Синтаксис 2**: f является функцией класса, а х

// является объектом или ссылкой на объект

p->f(); // **Синтаксис 3**: f является функцией класса,

// а р содержит указатель на х

Рассмотрим гипотетическую функцию, предназначенную для «проверки» объектов Widget:

void test(Widget& w); // Проверить объект w. Если объект не

// проходит проверку, он помечается как "плохой"

Допустим, у нас имеется контейнер объектов Widget:

vector<Widget> vw; // vw содержит объекты Widget

Для проверки всех объектов Widget в контейнере vw можно воспользоваться алгоритмом for\_each:

// **Вариант 1** (нормально компилируется)

for\_each(vw.begin(), vw.end(), test);

Но представьте, что test является функцией класса Widget, а не внешней функцией (то есть класс Widget сам обеспечивает проверку своих объектов):

class Widget

{ public:

…

void test(); // Выполнить самопроверку. Если проверка

… // завершается неудачей, объект помечается

}; // как "плохой"

В идеальном мире мы могли бы воспользоваться for\_each для вызова функции Widget::test всех объектов вектора vw:

for\_each(vw.begin(), vw.end(), &Widget::test);

// **Вариант 2** (не компилируется!)

Более того, если бы наш мир был действительно идеальным, алгоритм for\_each мог бы использоваться и для вызова Widget::test в контейнере указателей Widget\*:

list<Widget\*> lpw; // Список lpw содержит указатели

// на объекты Widget

// **Вариант 3** (не компилируется!)

for\_each(lpw.begin(), lpw.end(),&widget::test);

Но подумайте, что должно было бы происходить в этом идеальном мире. Внутри функции for\_each в варианте 1 вызывается внешняя функция, поэтому должен использоваться синтаксис 1. Внутри вызова for\_each в варианте 2 следовало бы использовать синтаксис 2, поскольку вызывается функция класса. А внутри функции for\_each в варианте 3 пришлось бы использовать синтаксис 3, поскольку речь идет о функции класса и указателе на объект. Таким образом, нам понадобились бы *три* разных версии for\_each — разве такой мир можно назвать идеальным?

В реальном мире существует только одна версия for\_each. Нетрудно представить себе возможную ее реализацию:

template<typename InputIterator, typename Function>

Function for\_each(InputIterator begin, InputIterator end,

Function f)

{ while (begin != end) **f(**\*begin++**)**;}

Жирный шрифт используется для выделения того, что при вызове for\_each используется синтаксис 1. В STL существует всеобщее правило, согласно которому функции и объекты функций всегда вызываются в первой синтаксической форме (как внешние функции). Становится понятно, почему вариант 1 компилируется, а варианты 2 и 3 не компилируются — алгоритмы STL (в том числе и for\_each) жестко закодированы на использование синтаксиса внешних функций, с которым совместим только вариант 1.

Теперь понятно, для чего нужны функции mem\_fun и mem\_fun\_ref. Они обеспечивают возможность вызова функций классов (обычно вызываемых в синтаксисе 2 и 3) при помощи синтаксиса 1. Принцип работы mem\_fun и mem\_fun\_ref прост, хотя для пущей ясности желательно рассмотреть объявление одной из этих функций. В действительности они представляют собой шаблоны функций, причем существует несколько вариантов mem\_fun и mem\_fun\_ref для разного количества параметров и наличия-отсутствия константности адаптируемых ими функций классов. Одного объявления вполне достаточно, чтобы разобраться в происходящем:

template<typename R, typename C> // Объявление mem\_fun

mem\_fun\_t<R, C> // для неконстантных функций без параметров;

**mem\_fun**(R(C::\*pmf)()); // С – класс; R - тип возвращаемого

// значения функции, на которую ссылается указатель

Функция mem\_fun создает указатель pmf на функцию класса и возвращает объект типа mem\_fun\_t. Тип представляет собой класс функтора, содержащий указатель на функцию и функцию operator(), которая по указателю вызывает функцию для объекта, переданного operator(). Например, в следующем фрагменте:

list<Widget\*> lpw;

…

for\_each(lpw.begin(), lpw.end(), **mem\_fun**(&Widget::test));

// Теперь нормально компилируется

При вызове for\_each передается объект типа mem\_fun\_t, содержащий указатель на Widget::test. Для каждого указателя Widget\* в lpw алгоритм for\_each «вызывает» объект mem\_fun\_t с использованием синтаксиса 1, а этот объект непосредственно вызывает Widget::test для указателя Widget\* с использованием синтаксиса 3.

В целом mem\_fun приводит синтаксис 3, необходимый для Widget::test при использовании с указателем Widget\*, к синтаксису 1, используемому алгоритмом for\_each. По вполне понятным причинам такие классы, как mem\_fun\_t, называются *адаптерами объектов функций.* Наверное, вы уже догадались, что по аналогии со всем, о чем говорилось ранее, функции mem\_fun\_ref адаптируют синтаксис 2 к синтаксису 1 и генерируют адаптеры типа mem\_fun\_ref\_t.

Объекты, создаваемые функциями mem\_fun и mem\_fun\_ref, не ограничиваются простой унификацией синтаксиса для компонентов STL. Они (а также объекты, создаваемые функцией ptr\_fun) также предоставляют важные определения типов. Тем не менее, стоит разобраться, почему конструкция

for\_each(vw.begin(), vw.end(), test); // См. ранее, вариант 1

// Нормально компилируется

компилируется, а следующие конструкции не компилируются:

for\_each(vw.begin(),vw.end(),&Widget::test); //См. ранее, вар.2

// Не компилируется.

for\_each(lpw.begin(),lpw.end(),&Widget::test);//См.ранее, вар.3

// Не компилируется

При первом вызове (вариант 1) передается настоящая функция, поэтому адаптация синтаксиса вызова для for\_each не нужна; алгоритм сам вызовет ее с правильным синтаксисом. Более того, for\_each не использует определения типов, добавляемые функцией ptr\_fun, поэтому при передаче test функция ptr\_fun не нужна. С другой стороны, добавленные определения не повредят, поэтому следующий фрагмент функционально эквивалентен приведенному выше:

for\_each(vw.begin(), vw.end(), **ptr\_fun**(test));

// Компилируется и работает, как вариант 1.

Если вы забываете, когда функция ptr\_fun обязательна, а в каких случаях без нее можно обойтись, лучше используйте ее при всех передачах функций компонентам STL. STL игнорирует лишние вызовы, и они не отражаются на быстродействии программы. Возможно, во время чтения вашей программы кто-нибудь удивленно поднимет брови при виде лишнего вызова ptr\_fun.

Существует и другой подход — использовать ptr\_fun в случае крайней необходимости. Если функция отсутствует там, где необходимы определения типов, компилятор выдает сообщение об ошибке. Тогда вы возвращаетесь к программе и включаете в нее пропущенный вызов.

С mem\_fun и mem\_fun\_ref ситуация принципиально иная. Эти функции всегда должны применяться при передаче функции компонентам STL, поскольку помимо определения типов (необходимых или нет) они адаптируют синтаксис вызова, который обычно используется для функций класса, к синтаксису, принятому в STL. Если не использовать эти функции при передаче указателей на функции класса, программа не будет компилироваться.

Остается лишь разобраться со странными именами адаптеров. Перед нами самый настоящий пережиток прошлого STL. Когда впервые возникла необходимость в адаптерах, разработчики STL ориентировались на контейнеры указателей (не стоит забывать, что контейнеры указателей поддерживают полиморфизм, а контейнеры объектов — нет). Когда понадобился адаптер для функций классов (MEMber FUNctions), его назвали mem\_fun. Только позднее разработчики поняли, что для контейнеров объектов понадобится другой адаптер, и для этой цели изобрели имя mem\_fun\_ref.

### Следите за тем, чтобы конструкция less<T> означала operator<

Допустим, объект класса Widget обладает атрибутами weight и maxSpeed:

class Widget

{public:

…

size\_t weight() const;

size\_t maxSpeed() const;

…

}

Будем считать, что естественная сортировка объектов Widget осуществляется по атрибуту weight, что отражено в операторе < класса Widget:

bool operator<(const Widget& lhs, const Widget& rhs)

{ return lhs.weight()<rhs.weight();

}

Предположим, потребовалось создать контейнер multiset<Widget>, в котором объекты Widget отсортированы по атрибуту maxSpeed. Известно, что для контейнера multiset<Widget> используется функция сравнения less<Widget>, которая по умолчанию вызывает функцию operator< класса Widget. Может показаться, что единственный способ сортировки multiset<Widget> по атрибуту maxSpeed основан на разрыве связи между less<Widget> и operator< и специализации less<Widget> на сравнении атрибута maxSpeed:

template<> // Специализация std::less

struct std::less<Widget>: // для Widget: такой подход

public // считается крайне нежелательным!

std::binаry\_function<Widget, Widget**,**  bool>

{ bool operator() (const Widget& lhs, const Widget& rhs) const

{ return lhs.maxSpeed() < rhs.maxSpeed(); }

};

Поступать подобным образом не рекомендуется. Многие программисты обращают внимание на то, что в приведенном фрагменте специализируется не обычный шаблон, а шаблон из пространства имен std.

Вообще говоря, попытки модификации компонентов std действительно запрещены, поскольку их последствия могут оказаться непредсказуемыми, но в некоторых ситуациях минимальные изменения все же разрешены. А именно, программистам разрешается специализировать шаблоны std для пользовательских типов. Почти всегда существуют альтернативные решения, но в отдельных случаях такой подход вполне разумен. Например, разработчики классов умных указателей часто хотят, чтобы их классы при сортировке вели себя как встроенные указатели, поэтому специализация std::less для типов умных указателей встречается не так уж редко. Далее приведен фрагмент класса shared\_ptr из библиотеки Boost:

namespace std

{template<typename T> // Специализация std::less

struct less<boost::shared\_ptr<T> >: // для boost::shared\_ptr<T>

public // (boost - пространство имен)

binary\_function<boost::shared\_ptr<T>, boost::shared\_ptr<T>,

bool>

{ bool operator()(const boost::shared\_ptr<T>& a,

const boost::shared\_ptr<T>& b) const

{return less<T\*>()(a.get(), b.get());}

// shared\_ptr::get возвращает встроенный указатель

// из объекта shared\_ptr

};

}

В данном примере специализация выглядит вполне разумно, поскольку специализация less всего лишь гарантирует, что порядок сортировки умных указателей будет совпадать с порядком сортировки их встроенных аналогов. К сожалению, наша специализация less для класса Widget преподносит неприятный сюрприз.

Программисты C++ часто опираются на предположения. Например, они предполагают, что копирующие конструкторы действительно копируют (невыполнение этого правила приводит к удивительным последствиям). Они предполагают, что в результате взятия адреса объекта вы получаете указатель на этот объект. Они предполагают, что адаптеры bind1st и not2 могут применяться к объектам функций. Они предполагают, что оператор + выполняет сложение (кроме объектов string, но знак «+» традиционно используется для выполнения конкатенации строк), что оператор - вычитает, а оператор == проверяет равенство. И еще они предполагают, что функция less эквивалентна operator<.

В действительности operator< представляет собой нечто большее, чем реализацию less по умолчанию — он соответствует *ожидаемому поведению* less. Если less вместо вызова operator< делает что-либо другое, это нарушает ожидания программистов. Конечно, поступать так не стоит — особенно если без этого можно обойтись.

В STL нет ни одного случая использования less, когда программисту бы не предоставлялась возможность задать другой критерий сравнения. Вернемся к исходному примеру с контейнером multiset<Widget>, упорядоченному по атрибуту maxSpeed. Задача решается просто: для выполнения нужного сравнения достаточно создать класс функтора практически с любым именем, *кроме* less. Пример:

struct MaxSpeedCompare:

public binary\_function<Widget, Widget, bool>

{bool operator()(const Widget& lhs, const Widget& rhs) const

{ return lhs.maxSpeed() < rhs.maxSpeed(); }

};

При создании контейнера multiset достаточно указать тип сравнения MaxSpeedCompare, тем самым переопределяя тип сравнения по умолчанию (less<Widget>):

multiset<Widget, MaxSpeedCompare> widgets;

Смысл этой команды абсолютно очевиден: мы создаем контейнер multiset с элементами Widget, упорядоченными в соответствии с классом функтора MaxSpeedCompare. Сравним со следующим объявлением:

multiset<Widget> widgets;

В нем создается контейнер multiset объектов Widget, упорядоченных по стандартному критерию. Строго говоря, упорядочение производится по критерию less<Widget>, но большинство программистов будет полагать, что сортировка производится функцией operator<. Не нужно подменять определение less. Если вы хотите использовать less (явно или косвенно), проследите за тем, чтобы этот критерий был эквивалентен operator<. Если объекты должны сортироваться по другому критерию, создайте специальный класс функтора и назовите его как-нибудь иначе.

# Примеры функторов

***Пример 1.***

#include <iostream>

#include <string>

using namespace std;

class TestFunctor

{ public:

// Перегруженный оператор вызова функции, принимает в качестве параметра

// объект класса string и не возвращает никакого значения

void operator() (const string& str) { cout << str << endl; }

};

int main (int argc, char\*\* argv)

{ **TestFunctor print;** // Создание функционального объекта

string str("C++ string");

**print(str);** // Вызов перегруженного operator()

**print("C string");** // Обратим внимание, что в качестве параметра

// передается не объект класса string, а С-строка; в данном случае будет неявно

// вызван конструктор класса string, который создаст на основе

// переданной С-строки временный объекта класса string

return0;

}

Вывод программы:

C++ string

C string

Класс функтора, т.е. класса с перегруженным оператором operator() является полноценным классом и может содержать конструкторы, дополнительные методы, а также внутренние данные. Кроме того, как и обычная функция operator() может быть перегружен, чтобы принимать различное количество аргументов и возвращать значение какого-либо типа. Ниже приведен пример функтора, для которого перегружено несколько версий operator(), принимающих различное количество аргументов.

***Пример 2.***

#include <iostream>

using namespace std;

struct Summ

{ int operator() (int arg1, int arg2) { return arg1 + arg2; }

int operator() (int arg1, int arg2, int arg3) { return arg1 + arg2 + arg3; }

int operator() (int arg1, int arg2, int arg3, int arg4) { return arg1 + arg2 + arg3 + arg4; }

};

int main (int argc, char\*\* argv)

{ Summ summ;

cout << "1 + 2 = " << summ(1,2) << endl;

cout << "1 + 2 + 3 = " << summ(1,2,3) << endl;

cout << "1 + 2 + 3 + 4 = " << summ(1,2,3,4) << endl;

return 0;

}

Вывод программы:

1 + 2 = 3

1 + 2 + 3 = 6

1 + 2 + 3 + 4 = 10

Возникает вопросы. В чем принципиальная разница между описанными функторами и обычными функциями? Имеют ли функторы преимущества? Когда имеет смысл их применять?

Рассмотрим следующий пример: необходимо написать функцию, которая бы при каждом вызове генерировала число равное степень двойки от 0 до 15, при этом при каждом последующем вызове генерировалась бы следующая степень двойки, а при достижении 15 степени, все бы повторялось сначала. Т.е. при первом вызове функция должна возвратить число 1 (20), при втором 2 (21), при третьем 4 (23), и т.д. до (215), а затем опять 1 (20). Очевидно, что для решения этой задачи функция должна сохранять некое состояние между своими вызовами. Состояние может быть сохранение в глобальной или статической переменное (сохранить состояние в автоматической переменной невозможно, так как они уничтожаются между вызовами функции). С помощью статических переменных эта задача может быть решена следующим образом:

***Пример 3.***

int generate()

{ static int power = 0;

if (power > 15) power= 0;

return(1 <<power++);

}

Функция генерирует числа в соответствии с заданными ограничениями, однако она имеет существенный недостаток. В определенный момент времени у функции может быть только один клиент, который вызывает ее для получения следующей степеней двойки, если функция будет вызываться из различных потоков, или из одного потока различными клиентами то результат ее работы будет отличаться от требуемого. Продемонстрируем это на примере: два клиента одновременно пытаются получить описанные числа:

int main (int argc, char\*\* argv)

{ int i;

for (i = 0; i < 5; i++)

{ cout << "[client 1]" << generate() << endl;

cout << "[client 2]" << generate() << endl;

}

return 0;

}

Вывод программы будет иметь вид:

[client 1]1

[client 2]2

[client 1]4

[client 2]8

[client 1]16

[client 2]32

[client 1]64

[client 2]128

[client 1]256

[client 2]512

В то время как каждый клиент ожидал получить правильную последовательность, и правильным выводом программы должно быть:

[client 1]1

[client 2]1

[client 1]2

[client 2]2

[client 1]4

[client 2]4

[client 1]8

[client 2]8

[client 1]16

[client 2]16

Дело в том, что функция generate может сохранять только одну копию своего состояния, и каждый вызов изменяет его. При использовании статических переменных принципиально невозможно организовать правильную работу это функции в случае одновременного использования несколькими клиентами.

При использовании функторов эту задачу можно решить следующим образом:

class TwoPowerGenrator

{ public:

TwoPowerGenrator() : power(0) {}

void reset() { power = 0; }

int operator() ()

{ if (power > 15)

power = 0;

return (1 << power++);

}

private:

int power;

};

int main (int argc, char\*\* argv)

{ TwoPowerGenrator generate1, generate2;

int i;

for (i = 0; i < 5; i++)

{ cout << "[client 1]" << generate1() << endl;

cout << "[client 2]" << generate2() << endl;

}

return 0;

}

В класс включен дополнительный метод, который позволяет осуществлять «сброс» функтора, что приводит к тому, что он при следующем вызове он начинает генерировать последовательность с начала.

Может возникнуть вопрос, для чего необходимо использовать перегрузку operator(), если вместо этого можно включить в класс метод (например, с именем generate\_next) и вызывать его.

Т.е. вместо:

TwoPowerGenrator generator1;

int next = generator1();

написать

TwoPowerGenrator gen;

int next = gen.generate\_next();

Ответ на это вопрос связан с использованием обобщенного программирования и шаблонов С++. Вспомним стандартный алгоритм generate, который заполняет последовательность значениями сгенерированными функцией или функтором gen.

template <class ForwardIterator, class Generator>

void generate(ForwardIterator first, ForwardIterator last, Generator gen);

Обобщенная (шаблонная) реализация алгоритма, принимает в качестве gen, нечто, для чего применима операция вызова функции, т.е. запись gen(). В качестве gen может быть использован как функтор, так и обычная функция, так как в обоих случаях применим оператор вызова функции. В случае использования специального метода, такого как generate\_next, пришлось бы иметь по крайне мере две реализации алгоритма generate, одну для обычных функций, вторую для методов класса, при этом, было бы необходимо передавать в качестве параметра указатель на член класса или использовать жестко заданное имя, что может быть неудобно.

# Функторы совместимые с STL

Функторы широко используются вместе со стандартными алгоритмами STL, однако чтобы функтор был совместим с STL, необходимо, чтобы он удовлетворял некоторым требованиям. Совместимый с STL функтор должен определять следующие имена типов:

*result\_type*– тип возвращаемого значения.

*argument\_type*– тип аргумента (в случае функтора принимающего один аргумент).

*first\_argument\_type*– тип первого аргумента (в случае функтора принимающего два аргумента).

*second\_argument\_type*– тип второго аргумента (в случае функтора принимающего два аргумента).

STLсодержит вспомогательные классы, которые могут быть использованы в качестве базовых в случае создания совместимых с STL функторов:

template<class Arg, class Res>

struct unary\_function

{ typedef Arg argument\_type;

typedef Res result\_type;

};

template<class Arg1, class Arg2, class Res>

struct binary\_function

{ typedef Arg1 first\_argument\_type;

typedef Arg2 second\_argument\_type;

typedef Res result\_type;

};

unary\_function должен быть использован в случае создания совместимого с STL функтора принимающего один аргумент, binary\_function в случае функтора принимающего два аргумента. Вспомогательные базовые классы должны быть использованы следующим образом:

#include <iostream>

#include <functional>

#include <algorithm>

#include <vector>

using namespace std;

template <class T>

void print\_to\_cout(const T& arg)

{ cout << arg << ',' ; }

class is\_even\_predicate: public unary\_function<int, bool>

{ public:

**bool** operator() (**int** arg)

{ return (arg % 2) == 0; }

};

int main (int argc, char\*\* argv)

{ vector<int> v1;

int i;

for (i = 0; i < 10; i++) v1.push\_back(i);

for\_each(v1.begin(),v1.end(), print\_to\_cout<int>);

cout << endl;

v1.erase(remove\_if(v1.begin(),v1.end(),is\_even\_predicate()),v1.end());

for\_each(v1.begin(),v1.end(), print\_to\_cout<int>);

cout << endl;

return 0;

}

В случае двухаргументного функтора, в качестве базового класса должна использоваться binary\_function

class add: public binary\_function<int, int, int>

{ public:

**int** operator() (**int** arg1,**int** arg1) { return arg1 + arg2; }

};

Все стандартные функторы STL являются наследниками unary\_function и binary\_function.

Определение базовых классов unary\_function и binary\_function, а также всех стандартных функторов находится в заголовочном файле <functional>

# Передача функтора в функцию в качестве аргумента

Необходимо учитывать, что в STL **функторы** передаются по значению. Дело в том, что в стандартной библиотеке указатели на обычные функции передаются в другие функции по значению, т.е. значение указателя на функцию копируется из вызывающей функции в вызываемую. Функторы STL создавались по образцу указателей на обычные функции (собственно все алгоритмы стандартной библиотеки могут принимать как функторы так и указатели на обычные функции), поэтому существует правило, согласно которому функторы передаются по значению.

В действительности есть способ передать функтор по ссылке, для этого явно указать параметры шаблона стандартного алгоритма, как это сделано в следующем примере:

#include <functional>

#include <algorithm>

#include <vector>

using namespace std;

class is\_even\_predicate: public unary\_function<int, bool>

{ public:

bool operator() (int arg) { return (arg % 2) == 0; }

};

int main (int argc, char\*\* argv)

{ vector<int> v1;

… // заполнить вектор

v1.erase(remove\_if<vector<int>::iterator, const is\_even\_predicate&>

(v1.begin(),v1.end(),is\_even\_predicate()),v1.end());

…

return 0;

}

На практике такой способ, несмотря на внешнюю привлекательность, не всегда используется, так как существует возможность, что некоторые алгоритмы не будут компилироваться, если функциональный объект передаётся по ссылке. Если известно, что алгоритм может принимать ссылку на функтор, то можно использовать этот способ, однако в общем случае следует предполагать, что функторы передаются по значению.

Исходя из того, что функторы передаются по значению, при их создании необходимо придерживаться двух принципов (по большей части это советы, но их соблюдение стало общепризнанной нормой программирования STL).

Функциональный объект должен быть небольшим, в противном случае передача его в качестве параметра по значению будет очень затратной.

Функтор не должен быть полиморфным. При передаче полиморфного объекта по значению происходит урезание объекта до базового класса.

Естественное, что не всегда функторы имеют небольшой размер, естественно также, что иногда необходимо обеспечить функтору полиморфное поведение. В этих случае может быть использован метод разделения функтора на две части: часть, которая передается алгоритмам по значению, и часть, которая содержит реальные данные функтора и обеспечивает полиморфное поведение.

Предположим, существует следующий полиморфный функтор, содержащий большой объем данных

template<class T>

class BigPolymorhicFunctor : public unary\_function<T, void>

{ public:

**virtual** voidoperator() (constT&val); // реализует полиморфное поведение

private:

**BigDataClass**data; // Класс содержит большой объем данных

};

Преобразуем класс следующим образом:

template<classT>

class BigPolymorhicFunctor : public unary\_function<T, void>

{ public:

. . .

void operator() (const T& val) { impl->operator()(val); }

private:

**BigPolymorphicFunctorImpl\* impl**; // Указатель на реальный объект-функтор

// содержащий данные и реализующий полиморфное поведение

};

template<classT>

class BigPolymorphicFunctorImpl

{ public:

. . .

**virtual** void operator() (const T& val); // Реализует полиморфное поведение

private:

BigDataClassdata;

};

Подобная реализация функтора позволит избежать копирования больших объемов данных и потери полиморфного поведения при передаче функтора BigPolymorhicFunctor по значению, однако необходимо понимать, что такая реализации приведет к существованию нескольких объектов BigPolymorhicFunctor, указывающих на один и тот же объект BigPolymorphicFunctorImpl со всеми вытекающими отсюда трудностями.

# *Варианты заданий*

***Задание 1***

Выполнить задачу в соответствии с номером варианта. При выполнении задачи использовать функциональные объекты (max – 4 балла).

***Задание 2***

Повторить выполнение своего варианта задания, но с использованием **параметризованных** **классов** (max – 5 баллов).

***Таблица 1***

|  |  |
| --- | --- |
| № | Задание |
| 11,15 | Дан целочисленный список. Используя функцию for\_each найти сумму простых элементов списка (по определению 0 и 1 простыми числами не являются). |
| 22, 16 | Дан символьный вектор. Используя функцию find\_if найти в векторе первое появление заданного символа. |
| 33, 17 | Дан список латинских символов. Используя функцию transform поменять регистр у каждой буквы списка. |
| 44, 18 | Дан целочисленный список. Используя функцию for\_each найти количество элементов списка, являющихся степенями тройки. |
| 55, 19 | Дан список строк. Используя функцию find\_if найти строку, первая буква которой является гласной. |
| 66, 20 | Дан целочисленный вектор. Используя функцию transform умножить на масштабирующий коэффициент все элементы вектора, не превышающие заданного числа. |
| 77, 21 | Дан целочисленный вектор. Используя функцию for\_each найти в векторе среднее геометрическое значений, являющихся точными квадратами. |
| 88, 22 | Дан целочисленный вектор. Используя функцию find\_if найти заданное с клавиатуры значение в этом векторе. |
| 99, 23 | Дан целочисленный список. Используя функцию transform умножить каждый элемент списка, являющийся степенью двойки, на масштабирующий коэффициент. |
| 110, 24 | Дан целочисленный список. Используя функцию for\_each найти сумму элементов списка, являющихся числами Фибоначчи. |
| 111, 25 | Дан список строк. Используя функцию find\_if найти первую строку-"перевертыш" т.е. такую, которую читать можно одинаково с конца и с начала, например, "казак" ,"шалаш". |
| 112, 26 | Дан целочисленный вектор. Используя функцию transform умножить на масштабирующий коэффициент все элементы вектора, являющиеся точными квадратами. |
| 113, 27 | Дан целочисленный вектор. Используя функцию for\_each найти в нем все числа-"перевертыши", т.е. такие, а которых порядок расположения цифр одинаков при чтении с конца и с начала, например, 5665, 12321. |
| 114, 28 | Дан целочисленный вектор. Используя функцию find\_if найти первое простое число (по определению 0 и 1 простыми числами не являются). |